Appears in the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software
Austin, Texas, March 10-12, 2004

Communication Breakdown: Analyzing CPU Usage in Commercial Web Workloads

Jaidev P. Patwardhan†, Alvin R. Lebeck†, and Daniel J. Sorin‡
{jaidev,alvy}@cs.duke.edu, sorin@ee.duke.edu.

†Department of Computer Science
Duke University
Durham, NC 27708

‡Department of Electrical and Computer Engineering
Duke University
Durham, NC 27708

Abstract

There is increasing concern among developers that future web servers running commercial workloads may be limited by network processing overhead in the CPU as 10Gb ethernet becomes prevalent. We analyze CPU usage of real hardware running popular commercial workloads, with an emphasis on identifying networking overhead. Contrary to much popular belief, our experiments show that network processing is unlikely to be a problem for workloads that perform significant data processing. For the dynamic web serving workloads we examine, networking overhead is negligible (3% or less), and data processing limits performance. However, for web servers that serve static content, networking processing can significantly impact performance (up to 25% of CPU cycles). With an analytical model, we calculate the maximum possible improvement in throughput due to protocol offload to be 50% for the static web workloads.

1 Introduction

Web servers are fast becoming critical components in the infrastructure of large businesses. Companies make significant investments to purchase, deploy and maintain these servers. To improve their performance, it is important to have a good understanding of the workloads these machines are likely to run. This knowledge can help identify current and future bottlenecks in the system and help direct research.

As the widespread availability of 10 gigabit ethernet nears, there is concern among developers that the networking stack may prove to be a bottleneck for commercial servers running web workloads. This concern is driven by the "1 Hz/1 bps" rule of thumb, which states that driving a line at 1bps requires a 1 Hz processor. This processing requirement has prompted an increasing amount of interest in network protocol offload into hardware [3,6,10,12,15].

This paper makes a contribution by providing an understanding of where CPU cycles are spent and the amount of network communication overhead for some important web workloads. By studying the CPU profiles of these workloads under a wide variety of loads, we identify both current and potential future bottlenecks in systems. In particular, we examine the network stack to see if it could be a limiting factor in terms of CPU overhead. We also use CPU profile data as input to an analytical model that bounds the potential benefit of protocol offload.

We deploy five web workloads (discussed in depth in Section 2): two that serve static web content and three that serve dynamic web content. The difference in the categories is in the amount of data processing done before responding to the user. The five workloads represent a broad range of applications deployed on the Internet. For our experimental methodology (Section 3), we run the workloads on real hardware—systems running the Linux operating system—and gather detailed CPU profile information.

Through our experiments (Section 4), we contradict the prevalent belief that as raw network performance increases, network processing will be the limiting factor for most web workloads. For the three dynamic web workloads we examine, the networking overhead is very small (3% or less of CPU cycles). For these workloads, data processing limits throughput. If data processing overheads do not reduce in the future, network processing optimizations (e.g., protocol offload) are unlikely to provide much benefit to this class of applications. In contrast, for the two static web workloads, there can be significant networking overhead (up to 25% of CPU cycles). We confirm and generalize these experimental results with a high-level analytical model (Section 5) that studies the effectiveness of protocol offload based on certain workload characteristics (e.g., ratio of application processing to network processing).

2 Commercial Web Workloads

Modern web servers run a variety of different workloads. Some serve simple static HTML documents, others serve dynamic content via Java, Perl, CGI, or some other active scripting language. Servers employed by commercial enterprises often utilize a multi-tier system, which includes Java (or .NET) based middleware and a database backend. We evaluate five workloads, representing both static web content (Section 2.1) and dynamic web content (Section 2.2)

2.1 Static Web Content

In this subsection, we describe two workloads that deliver static web content to users. Both workloads use the multithreaded version (2.0.40) of the popular open source Apache web server. The base configuration is modified to allow over 1000 simultaneous connections. Request logging is disabled to reduce server side overhead. This is a common procedure to improve performance by reducing disk I/O.

2.1.1 Apache Bench

Apache Bench (AB) is a micro-benchmark that stress tests the server installation and does not exercise memory or I/O. The program can be configured to set up a number of connections to the server and fetch a specified file a given number of times. In our experiments, we vary the size of the file fetched by AB from a small 509 byte file to a larger 64KB file. Small files do not require fragmentation, but may incur more frequent system calls and interrupts. The large files require fragmentation into network packets. In each experiment we use 500 clients requesting the same file.

2.1.2 Apache with Surge

The second static web content workload uses Surge [1] as the file and load generator. Surge creates a set of files using a zipf distribution to mimic web file sizes [4]. The Surge client models several users requesting files from the generated set. We use http 1.1, which uses persistent connections, thus each connection typically requests and receives more than one file.

By controlling the size of the file set generated by Surge, we ensure that the memory and I/O system of the target web server are exercised. The serving of static web content requires little, if any, data processing at the server side. It largely involves fetching the file from the I/O system and sending it across the network. This is the main distinguishing factor between the first two and the remaining workloads.

2.2 Dynamic Web Content

In this subsection, we describe three workloads that deliver dynamic web content. The first is based on an existing system that is very popular on the internet (slashdot.org), the second is an industry standard SPEC benchmark run on web servers while the third is a Java middleware workload.

2.2.1 Apache with SlashCode

The first dynamic web content workload is formed using Apache 1.3.27[1] with SlashCode [8] 2.2.7. SlashCode is the perl source code to the popular slashdot.org website. It uses perl scripts to generate dynamic web content from the backend database. We use MySQL v8.23 as the database backend to store user information, articles, and posts. We use a driver program that generates requests for articles from the main page of the website. The driver program mimics a number of clients accessing the system, and it uses a probabilistic state machine to control the next client action (read article, post article, respond to poll, etc.). The workload is a good example of serving a mix of static and dynamic content, and it is similar to the real world setup of the slashdot.org site.

1. Slash does not support Apache 2.0.40's version of mod_perl yet.

2.2.2 SPECweb99_SSL

SPECweb99_SSL [11] is the counterpart of the SPECweb99 benchmark from SPEC. It tests a web server serving a mix of dynamic and static content over an SSL connection. SPEC provides a specification that the target system must meet (especially for routines that serve the dynamic content). Our setup involves the use of a popular commercial web server[2] that provides C routines for common CGI operations. This workload differs from the previous one in two important ways. First, it uses SSL, which introduces additional overhead per connection. Second, it uses C based routines for generating the dynamic content. Since the C code is compiled, its execution will be more efficient than interpreted perl code. The use of a "standard" SPEC benchmark enables us to test our configuration with published results from SPEC, and it ensures that our system setup is similar to commercial systems. The benchmark client runs on one or more machines synchronized by a driver program. The total number of client threads is evenly distributed among the client machines. The driver uses a probabilistic state machine to determine the action to perform (static GET, dynamic GET, or POST). The dynamic content is generated by a custom ad generator on the server side.

2.2.3 Trade 3

The final workload that we use is IBM's open source Trade 3 [7] benchmark. Trade 3 models an online stock brokerage application. It provides a real world workload that utilizes the Enterprise Java Beans v2.0 (EJB) architecture. It also uses message driven beans, multi-phase database transactions, and web services. Object persistence is managed by the EJB containers. The application server used to run Trade3 is IBM's WebSphere 5.0, and we use IBM DB2 v7.1 as the database backend. The Trade3 application has a web interface that performs a random transaction when referenced. We test the performance of our setup using IBM's Web Performance Tools (WPT).

3 Experimental Goals and Methodology

In this section, we briefly discuss the goals of our experiments and then describe our methodology.

3.1 Experimental Goals

The primary goal of this work is to find current and future bottlenecks in web workloads. To achieve this high-level goal, we examine three aspects of the system.

First, we look at the total (user+kernel) CPU profile. This gives us an idea of where the system spends most of its time when running the current workload. It shows whether the system is spending more time running the application or the operating system kernel. Second, we examine the code segments in which the system spends the longest amount of

2. The web server remains unnamed for licensing reasons.

time. If it is in the application, it is important that we know the exact tasks the application performs. If the system spends most of the time in the kernel, we note the code segments where time is spent within the kernel and understand why time is being spent there. Third, we look for code segments that could form the bottleneck if the primary bottleneck is removed. For example, what if network processing is no longer a bottleneck?

3.2 Methodology

Choosing a methodology and target system for our experiments involved some trade-offs. We wanted to run real workloads for a significant amount of time and we wanted to avoid making simplifying assumptions about the target system, which precluded the use of simulation. We thus chose to use real hardware, with the obvious limitation that it cannot be configured to emulate an arbitrary number of possible target systems. Since comparing results across platforms is problematic, we chose a single, prevalent platform: Intel processors and the Linux operating system.

Our experimental setup consists of four machines connected by a Cisco Catalyst switch with gigabit ethernet links. The main system under test is a 1.4GHz Pentium III, with 1GB of RAM. All server software, including databases where applicable, executes on this machine. Client driver software executes on one or more of the other machines. One of the driver machines is a 2.2 GHz Pentium 4 with 2GB of RAM. The other two are 1.4GHz Pentium III's with 512MB RAM. The P4 system was not chosen as the main system under test due to incompatibility with some of our tools. All four machines run Debian Linux, with the 2.4 series kernels. All the machines have local SCSI disks, but the workloads run off an NFS file system. To gather CPU profile data, we used the Intel® VTune™ profiling tool. VTune uses processor performance counters to gather CPU profiling data.

We also tested three of our workloads on a multiprocessor system. For these tests, we used an SMP-enabled version of kernel version 2.4.20. The SMP system had dual P-III's running at 1GHz. The results (not shown due to space limitations) were similar to uniprocessor results, except for scheduling overhead. The dual processor SMP version of the kernel is known to have problems, and we saw these problems manifested as higher scheduling overheads.

All the workloads are warmed up before collecting CPU profile data. Table 1 lists the duration that each workload runs. CPU profile data is collected over a period of 20 seconds, with 20 seconds of calibration preceding the data collection. For all workloads, the data we present is the arithmetic mean over three runs. VTune has two components for profiling linux machines. The main VTune analyzer program runs on a remote windows machine. This communicates with the target system under test through the VTune Server. Data is gathered on the target system using a kernel loadable module. Since the analyzer itself does not run on the target system, interference due to measurement is minimized. The VTune server adds a small amount of overhead which is reported in the profiling data.

Workload	Duration
ApacheBench	>100,000 requests
Surge/Apache	120 seconds
Slash/Apache	≥ 500 object requests
SPECweb99_SSL	> 4 minutes
Trade3	200 seconds

TABLE 1. Workload Test Durations

4 Results and Analysis

This section describes our experiments and results in detail. We divide our discussion into two sections by workload type: static web and dynamic web. We provide a summary of the results in Section 4.3.

For all the workloads, we gather detailed CPU profile information. This information includes code segment names and the amount of CPU time spent within them. The code segments are divided into different categories, some of which are common across all workloads. We provide a brief explanation of the various category names reported in the CPU profile in Table 2.

4.1 Static Web Workloads

In this section, we describe the results of our experiments on AB and Surge.

4.1.1 Apache Bench

AB is the smallest of our workloads. We configure it to request the same file at least 100,000 times, with the requests distributed evenly among 500 client threads. We vary the file size to study its effect on the CPU profile as well as the throughput achieved.

Table 3 shows the results of AB for three file sizes. We see that increasing the file size increases the bandwidth utilization, but decreases the requests per second satisfied by the server. The bandwidth increases by 5x, but the throughput, in terms of requests per second, drops by a factor of 3. The increase in bandwidth is mainly because the larger files can better utilize the bandwidth of the network per request.

Figure 1 shows the user and kernel CPU profile and Figure 2 shows the zoomed in profile for the linux kernel. From Figure 1, we see that the benchmark regularly saturates the CPU on the server (no idle time). The most significant CPU fractions are due to the linux kernel, vmlinux (>40%), and apache httpd (>10%). Other significant fractions are due to the NIC driver, e1000, and the various libraries. Any category contributing less than 1% is grouped into "Misc." Within the kernel itself (Figure 2), the networking code occupies the largest fraction, varying from around 15% (for

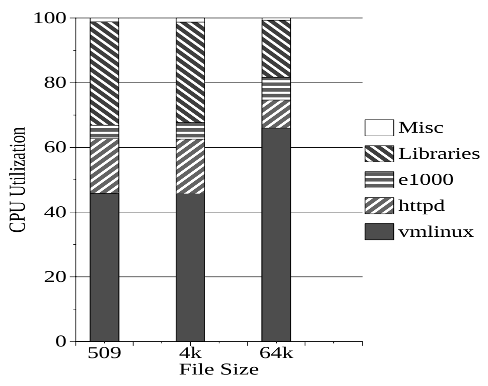

FIGURE 1. AB: CPU Utilization (Total)

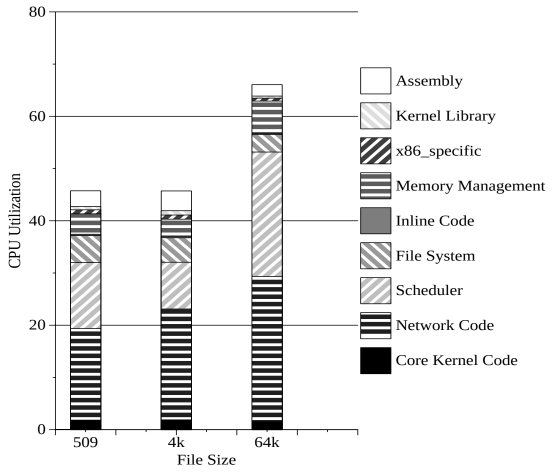

FIGURE 2. AB: CPU Utilization (Kernel)

Category	Explanation
vmlinux	Linux kernel
e1000	Intel E1000 Device Driver
bcm5700	Broadcom Gigabit NIC Driver
Libraries	Various libraries, including libc (no Java)
Idle	"Default Idle" routine
libjvm	Java libraries, libjvm, libjava, libjitc
Misc	Routines that use <1% total CPU cycles
httpd	Apache web server
Appserver	WebSphere 5.0
sunrpc	Sun RPC routines
nfs	Network File Service routines

TABLE 2. CPU Profile: Categories

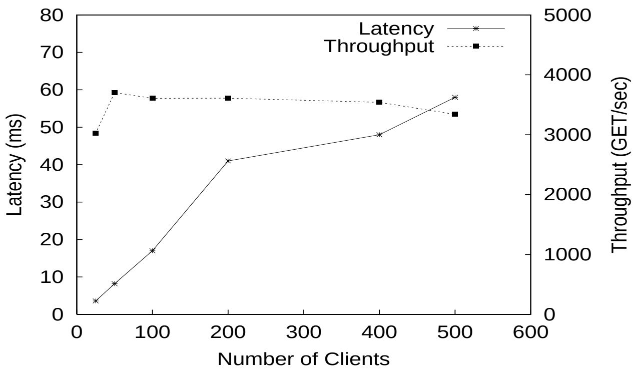

FIGURE 3. Surge: Throughput & Latency vs. # Clients

File Size	Total Time	Req/Sec	Throughput (Mbps)	Time per request (ms)	Requests
509 B	46.96s	4258.94	26.88	117.4	200,000
4KB	47.82s	4182.26	143.6	119.6	200,000
64KB	69.97s	1429.29	720.0	349.8	100,000

TABLE 3. Apache Bench: Client Side Run Results

the 509B file size) to almost 30% (for 64KB files) of total CPU time. The increase in networking overhead is due to the larger quantities of data being handled. In case of the 509B file, TCP bundles multiple packets together to fit into an ethernet packet. In case of the 64K file, the networking layer has to deal with fragmentation in addition to regular tasks. The large task scheduling overhead is because of the number of threads the web server spawns to deal with the incoming requests.

Summary. From the experiments, we see that the two factors that limit throughput for AB are the scheduling overhead (up to 25%) and network stack overhead (up to 25%).

4.1.2 Surge and Apache

This workload is setup by having Surge generate a 480MB file set (20000 files) using a zipf distribution. The parameters for the zipf distribution unmodified from their default values in Surge. The smallest file is 75 bytes, and the largest file is 6MB. The web document root resides on an 800MHz Pentium III, 1.7TB NFS server with RAID-5. The client program is configured to run for at least 2 minutes.

We vary the number of client threads from 25 to 500. Figure 3 shows the throughput and latency of the web server as a function of the number of clients. As the number of clients is increased, the time taken to service requests also increases. Throughput continues to increase with the number of clients, up until 200 clients. After that, the server saturates and the throughput decreases. We see a drop in the throughput and a rise in response time beyond 400 clients.

We examine the CPU profile of this workload to explain the throughput behavior. From the CPU profile (Figure 4) we see that as we increase the number of clients from 25 to 50, we see a big jump in the kernel overhead. Beyond that, the

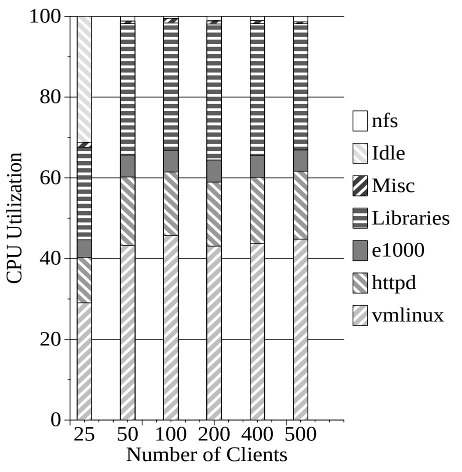

FIGURE 4. Surge: CPU Utilization (Total)

kernel overhead is almost constant. As we increase the number of clients, idle time drops (0% at 50+ clients), and web server and scheduling overhead increases. As the system saturates, throughput drops and latency increases. Figure 5 shows the breakdown within the linux kernel, where networking overhead remains the largest fraction. The network overhead increases as the number of clients increases from 25 to 50 then plateaus at 20% of overall CPU cycles.

Summary. For Surge, we find that the largest overhead is due to the networking stack (20% total CPU cycles), and that the Apache web server consumes a large fraction (~15% core + 25% of libraries) of the CPU.

4.2 Dynamic Web Workloads

In this section, we examine the results of our experiments on the dynamic web workloads. We analyze results from SPECweb, SlashCode and Trade3.

4.2.1 SlashCode

After performing a regular install of slashcode, we populate its database with random entries using tools provided in the distribution. In each run, we keep the number of objects fetched constant (500). Each object consists of multiple files (images, data). We start the test with a single user (and 500 requests), and increase the number of users to 50 (with 10 requests). Each test takes on the order of a minute. In all cases, the system is warmed up by running the site and the driver for 2000 requests. Figure 6 shows the throughput as a function of the number of clients.

We see that the throughput increases as we add more users, until the server saturates and then it decreases. The explanation can be found by examining the CPU profile graph (Figure 7) which shows decreasing idle time with increasing users. One important difference between this workload and the previous two is the fact that the dominant overhead is not the linux kernel. In fact, the kernel utilizes less than 5% of the CPU in all cases. Most of the overhead is due to perl processing within apache. This can be seen by examining a breakdown of httpd, where 75% or more time is spent in perl processing. There is little communication overhead, which is expected for such low throughput (~2Mbps).

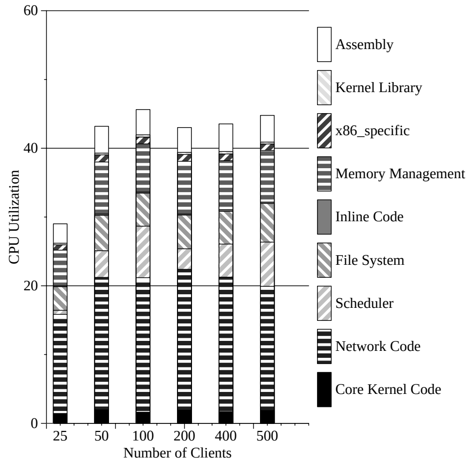

FIGURE 5. Surge: CPU Utilization (Kernel)

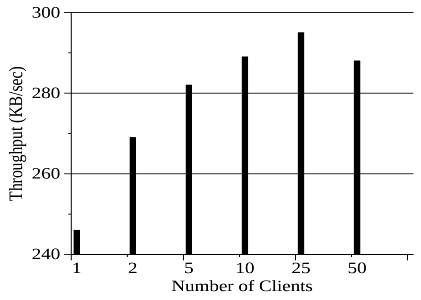

FIGURE 6. SlashCode: Throughput vs. # of Clients

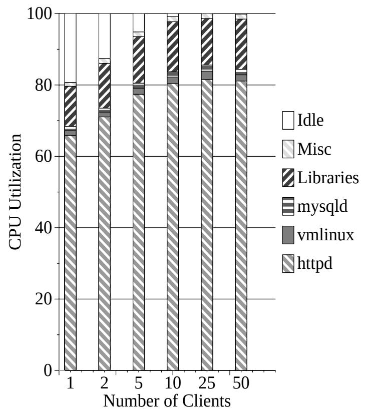

FIGURE 7. SlashCode: Utilization vs. # of Clients

Summary. The main factor limiting throughput for SlashCode is perl processing (up to 56% of total CPU time). The networking overhead is almost negligible. Thus, we expect little benefit for this workload through protocol offload. The largest benefit would be through optimized perl processing.

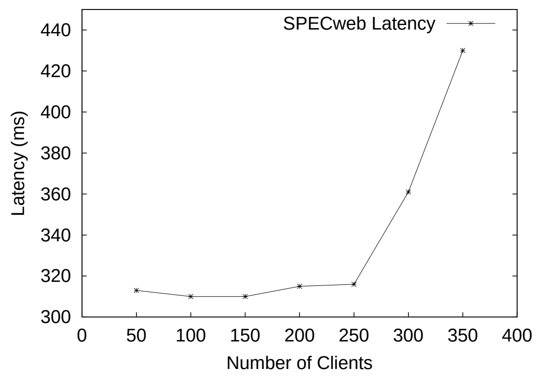

FIGURE 8. SPECweb: Latency vs. # of Clients

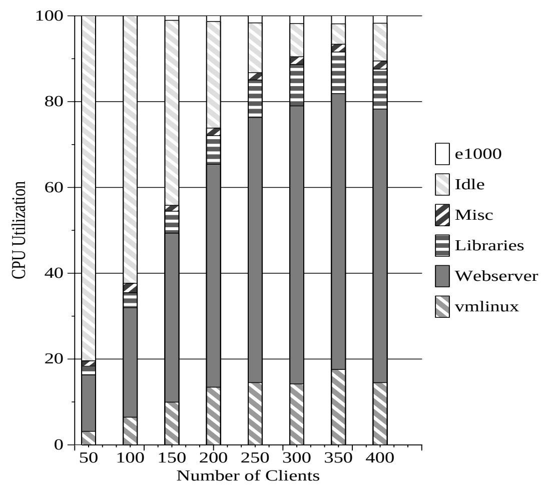

FIGURE 9. SPECweb: CPU Utilization (Total)

4.2.2 SPECweb99_SSL

SPECweb99_SSL uses a file generator to create its file set. This utility creates a large number of directories, each being about 5MB. The number of directories depends on the load to be applied during the benchmark run. A load of 500 (simultaneous connections) generates a 1.5GB file set.

For most data gathering runs, we run the benchmark for 4 minutes with 40 seconds of warm-up. We also do a full SPEC run to ensure that we have a reasonable configuration competitive with previously submitted SPEC results. A standard SPEC run must satisfy some requirements, including a minimum warm-up time, a minimum number of runs, a minimum bandwidth reported by all clients, and a maximum number of errors per client. We gather data for 50 to 350 simultaneous connections. Without tuning, we perform a SPEC compliant run (unpublished) with 225 connections.

Figure 8 shows the average time per request over all connections. There is little increase in the latency of the system until we have 250 clients. After that, we see a sharp rise in the latency. This happens as the system saturates. Figure 9 shows the CPU profile for specweb. We see that for 50 clients, the server has plenty of idle time (80%). As we increase the number of clients, the idle time drops quickly. It never reaches zero because of issues on the client side. We are able to saturate the machine if running a non-specweb driver. We see more kernel overhead here compared to SlashCode. Other than this, the two results are similar, with the web server consuming the largest fraction.

Summary. SPECweb overheads are dominated by the web server (~50% of CPU time) that generates dynamic content by executing cgi scripts. Network stack overhead is low and thus we expect little benefit from protocol offload.

4.2.3 Trade3

Trade3 is a Java middleware workload that models an online stock brokerage application. The Trade3 database is populated with 500 users, each of whom owns a set of shares and trades them through the application. Trade3 includes a servlet which performs a random trade when invoked. Thus, by repeated invocations of this servlet, we simulate a server running a multi-tier system for trading stocks. We vary the number of clients connecting to the server from 2 to 40.

The results, shown inFigure 10, show that the server can handle a relatively constant number of requests per second, but increasing the number of clients increases the latency per request. Once we reach 40 clients, the number of pages served per second drops. If we examine the CPU profile in Figure 11, we see that most of the overhead is due to the application server and Java libraries. The linux kernel occupies less than 10% of total CPU cycles.

Summary. Trade3 is another dynamic web workload with low network overhead. CPU time is dominated by Java processing (~25%) and the application server (~30%). We again expect little benefit from protocol offload.

4.3 Results Summary

Our experiments show that, for static web workloads, there is significant overhead due to thread management (up to 25%) and the network stack (up to 30%). For dynamic web workloads, however, the data processing time dominates all other tasks. The throughput of the these workloads(~10 Mbps) is much lower than the static web workloads (~100Mbps). To improve throughput for these workloads, we must target the primary bottlenecks.

To reduce data processing overheads, we can try and optimize the code that performs the data processing, or optimize the compiler/interpreter used to run that code. To reduce the network stack overhead, we can offload the protocol stack into hardware. This would reduce CPU load.

5 Analytical Model

In this section, we present an analysis of the potential benefits of protocol offload for our workloads using a high-level analytical model developed by Shivam and Chase [9]. This model uses four ratios—lag ratio, application ratio, wire ratio, and structural ratio—to identify system bottlenecks

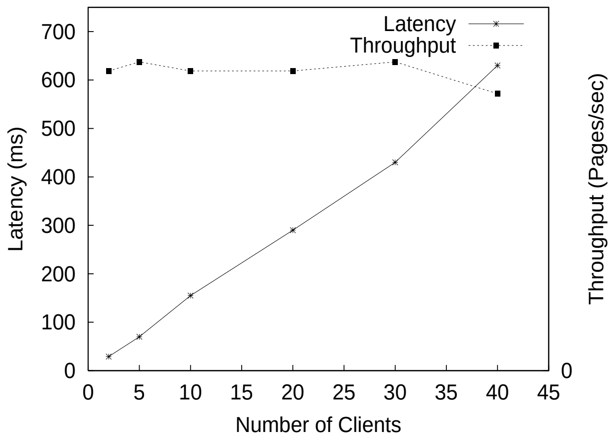

FIGURE 10. Trade3: Latency & Throughput vs. # of Clients

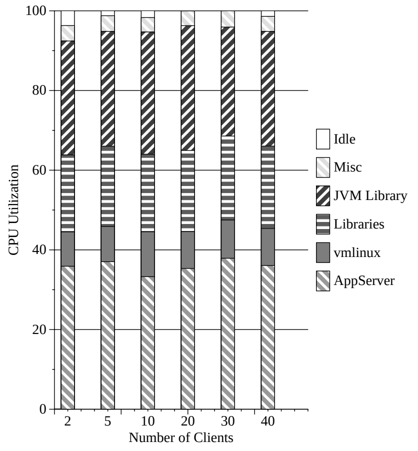

FIGURE 11. Trade3: CPU Utilization (Total)

Ratio	Explanation
Lag (α)	Ratio of Processor to NIC speed
Application (γ)	Ratio of normalized application processing to communication processing
Wire (σ)	Fraction of network bandwidth NIC can achieve
Structural (β)	Fraction of networking overhead off-loaded to NIC

TABLE 4. Model Ratios

Workload	Network Overhead	Gamma (γ)	Max Improvement in Throughput
AB	34%	1.95	52%
Surge	25.6%	3	34.5%
SlashCode	1%	99	1%
SPECweb	6.5%	14	7%
Trade3	4%	24	4%

TABLE 5. Benefits of Protocol Offload

and estimate the benefits of protocol offload. Table 4 describes these ratios. For the purposes of this work, the application ratio is most important.

The application ratio (γ) is the ratio of normalized application processing to communication processing. To calculate the application ratio, the model needs two input parameters, the CPU occupancy for communication overhead per unit of bandwidth, normalized to a reference host (o), and the CPU occupancy for application processing per unit of bandwidth normalized to a reference host (a). Thus, $\gamma = a/o$. For the rest of this discussion, we assume that the remaining ratios (α, β, σ) are 1. This implies that the NIC is not the bottleneck. We make this assumption as we are targeting future network hardware with plentiful bandwidth.

The application ratio puts a limit on the maximum possible benefits that could be derived by protocol offload in an ideal case. The maximum benefit achievable is a doubling of throughput, when $\gamma=1$. This is limited by $p/(\gamma-p+1)$, where p is the fraction of network overhead that can be off-loaded to the NIC. If we assume p=1 (ideal case), then the maximum benefit is bounded by $1/\gamma$. This assumes that the network card is itself not limited in any way due to the protocol processing that it now has to perform (since all other ratios are assumed to be 1).

We now examine our applications to see how much each could possibly benefit by protocol offload. The results presented do not reflect on the larger scope of protocol offload, but are applicable to servers running similar workloads.

5.1 Results

We now look at the results obtained for the three types of workloads. We expect maximum benefits for workloads that perform little data processing. Table 5 summarizes the results from the analytical model.

Static Web Content. The results for static web content workloads revel a significant fraction of network processing overhead. In the case of AB, the network stack occupies at most 27% of CPU cycles. In addition, the NIC driver occupies an additional 7%, making it a total of 34% network processing overhead. Assuming an ideal scenario in which all this overhead could be off-loaded, the maximum improvement achievable is $1/\gamma$. In reality, there would still be some interfacing overhead. Here, γ is 1.95. Thus, the maximum increase we would see in throughput in the case of ideal protocol offload is 52%. For Surge and Apache, the network stack occupies at most 20% of CPU cycles, and the NIC driver occupies a further 5.6%. This gives a γ of 3, which bounds the maximum increase in throughput at 34.5%.

Dynamic Web Content. In the dynamic content workloads, network stack processing overhead is small. For slashcode, the total overhead is at most 1%, giving a total possible benefit of 1%. For SPECweb, the overhead is higher, at 6.5% leading to a potential peak benefit of 7%.

Trade3. The result for Trade3 is similar. The total networking overhead is at most 4%, which leads to a maximum increase in throughput of 4%.

5.2 Conclusions from Model

Results from the analytical model show that protocol offload can potentially improve throughput, but only for applications in which networking overhead occupies a significant fraction of CPU cycles. For our static web workloads, the model predicts up to a 50% improvement in throughput if we use protocol offload. However, most assumptions made by the model are optimistic in favor of protocol offload. Due to interfacing overheads, the actual benefits are likely to be lower that those predicted by the model. This precludes improvements that would take place if the data processing overheads were to reduce.

Consider the significant perl processing overhead in SlashCode. Assume reducing the perl overhead by 50% translates directly to an improvement in bandwidth utilization (i.e., all CPU cycles freed by perl are taken up by the network stack). In this scenario, offloading the network stack to hardware could achieve a 62.5% improvement in throughput, compared to 1% without the improved perl processing.

6 Related Work

Extensive prior work has examined commercial workloads. We focus here on analyses that explore network processing. Researchers have proactively pursued protocol offload [3,6,10,12,15], as well as its feasibility, in anticipation of increased network bandwidth. Recent work has also analyzed the TCP stack in modern linux kernels and re-validated the old "1 Hz/1bps" rule of thumb for TCP/IP processing [5]. Other researchers have examined techniques to reduce data movement overhead by caching data on the network interface [13,14]. Binkert et al. [2] present a full system simulator targeted at network intensive applications and examine the memory system behavior of SPECweb99. Our work differs from the above studies by examining a broad class of workloads that includes sophisticated commercial workloads with significant data processing requirements.

7 Conclusions

To improve the performance of web servers, it is important that we understand the workloads they run. This paper presents an analysis of commercial web workloads in terms of their CPU profile and networking overheads. We show that, except for static web content serving, CPU usage is dominated by data processing overheads for generating dynamic content. Thus, if data processing overheads do not reduce in the future, protocol offload is unlikely to provide much benefit to this class of applications. However, for workloads that only serve data, the overhead due to the network stack can be significant. For static web content serving workloads, this can be as high as 50%. Protocol offload may provide significant benefits to similar workloads, such as storage area networks.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants No. CCR-0208920, EIA-9972879, Intel and equipment donations from IBM and Intel. We would like to thank David Becker, Jeff Chase, and Piyush Shivam for their help and useful feedback on our work. Sorin is supported by a Duke University Warren Faculty Scholarship.

References

[1] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[2] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full System Simulation using M5. In *Proceedings of the Sixth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, February 2003.

[3] Philip Buonadonna and David Culler. Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 247–256, May 2002.

[4] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-based Traces. Technical Report TR-95-010, Boston University, June 1995.

[5] Annie P. Foong, Thomas Huff, Herbert J. Hum, Jaidev P. Patwardhan, and Greg Regnier. TCP Performance Re-Visited. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 70–79, March 2003.

[6] The iReady and National Semiconductor Partnership. Offload Whitepaper. http://www.national.com/appinfo/networks/files/whitepaper1.pdf, 2003.

[7] Tony Lau and Yongli An. Running WebSphere Benchmark Sample Trade3 with Web Performance Tool (WPT). IBM Developer Works Online Library, March 2003.

[8] Open Source Developer Network (OSDN). SlashCode. http://slashcode.com.

[9] Piyush Shivam and Jeffrey S. Chase. On The Elusive Benefits of Protocol Offload. In *Proceedings of Workshop on Network-I/O Convergence: Experience, Lessons and Implications (NICELI)*, August 2003.

[10] Piyush Shivam, Pete Wyckoff, and Dhabaleshwar Panda. OS-Bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of SC2001*, November 2001.

[11] SPEC. SPEC WEB99_SSL. http://www.spec.org/web99ssl.

[12] Eric Yeh, Herman Chao, Venu Mannem, Joe Gervais, and Bradley Booth. Introduction to TCP/IP Offload Engine (TOE), April 2002.

[13] Kenneth G. Yocum, Jeffrey S. Chase, and Amin Vahdat. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[14] Hyong-youb Kim, Vijay Pai, and Scott Rixner. Increasing

Web Server Throughput with Network Interface Data Caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 239–250, October 2002.

[15] Hyong-youb Kim, Vijay Pai, and Scott Rixner. Exploiting Task-Level Concurrency in a Programmable Network Interface. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 2003.