Performance Scalability of a Multi-Core Web Server

Bryan Veal and Annie Foong
Intel Corporation
2111 NE 25th Ave
Hillsboro, OR 97124
{bryan.e.veal,annie.foong}@intel.com

ABSTRACT

Today's large multi-core Internet servers support thousands of concurrent connections or flows. The computation ability of future server platforms will depend on increasing numbers of cores. The key to ensure that performance scales with cores is to ensure that systems software and hardware are designed to fully exploit the parallelism that is inherent in independent network flows. This paper identifies the major bottlenecks to scalability for a reference server workload on a commercial server platform. However, performance scaling on commercial web servers has proven elusive. We determined that on web server running a modified SPECweb2005 Support workload, throughput scales only 4.8× on eight cores. Our results show that the operating system, TCP/IP stack, and application exploited flow-level parallelism well with few exceptions, and that load imbalance and shared cache affected performance little. Having eliminated these potential bottlenecks, we determined that performance scaling was limited by the capacity of the address bus, which became saturated on all eight cores. If this key obstacle is addressed, commercial web server and systems software are well-positioned to scale to a large number of cores.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance Attributes

General Terms

Design, Measurement, Performance.

Keywords

Scalability, parallelism, networks, web servers, cache hierarchies, network protocol stacks, load balancing.

1. INTRODUCTION

Today's Internet servers are designed to support thousands of concurrent client connections. Future general purpose computing platforms are shaped by the technology trends of chip-multiprocessing, multi-gigabit Ethernet networking, and large-capacity storage. As such, Internet servers will be built with highly-resourced, multi-core systems, and they will be expected to support increasing numbers of connections.

The key to Internet server performance lies in the ability to scale with hardware resources on a single system. Clients communicate with servers through mutually independent *flows* (or connections). If Internet server application processing and the associated network protocol processing of a flow are done exclusively on a single core, we expect minimal data sharing and synchronization between flows. We therefore expect that systems and application software can exploit this *flow-level parallelism* to scale with the number of cores.

Web servers continue to be among the most popular Internet applications [1]. Since they respond to mutually independent client requests, they should scale easily with the number of cores by exploiting flow-level parallelism. To test this, we set up test server running a well-tuned Apache HTTP server and Linux operating system. The server had eight cores with pairs of cores sharing L2 cache. The Linux kernel supports a parallelized network stack and the Apache web server is multi-threaded with one thread per connection.

Despite inherent parallelism and advances in system design, web server performance scaling has proven elusive. Our experiments showed that the test server running a modified SPECweb2005 Support workload achieved only a 4.8× speedup in throughput (compared to the ideal 8×). The number of cycles executed per byte of data transmitted by the server increased from 24.9 cycles/byte on one core to 44.2 cycles/byte on eight cores. Function-level profiling revealed that memory stalls were primarily responsible poor scaling.

Official SPECweb2005 results show similar scaling problems. For the Support workload, an eight-core AMD Opteron system [24] scales 1.5× over a comparably provisioned four-core system [22] for the number of concurrent sessions (analogous to throughput). Likewise, a sixteen-core Intel Xeon system [23] scales 1.3× over a comparable 8-core system [21] for the Support workload, and the 8-core system scales 1.6× over a comparable 4-core system [25]. Scaling of the other SPECweb2005 workloads correlates closely to the Support workload on these systems.

Precisely pinpointing the causes of poor performance scaling on a multi-core hardware system running a commercial server workload is non-trivial. We arrived at insights based on careful design of experiments and extensive code walk-

ANCS'07, December 3–4, 2007, Orlando, Florida, USA.

through of the application and systems software. The contribution of this work is therefore twofold:

1. we provide insights on the key causes of poor scalability of a web server, which ensure that optimizations focus on the correct bottlenecks and thus lead to performance improvements; and

2. we provide the analysis methodology leading to these insights.

Our analysis definitively ruled out several potential causes for poor scaling. We determined that, when flows are affinitized, the network protocol stack presented little contention between cores and its performance scaled linearly with the number of cores. Adding the web server application, we found negligible overheads in load balancing or core-to-core thread migration, confirming that the Linux scheduler handled SPECweb2005 workload adequately. We further confirmed that L2 cache and DTLB misses per flow did not increase significantly per unit of work when the number of cores was increased.

In general, we found the kernel and application to be programmed in a scalable manner. We found two instances where the system executed more instructions per unit of work as the number of cores was increased. The filesystem suffered additional overheads from spin locks for the directory cache, which was shared between all the cores. The web server application suffered from taking more steps to traverse over-filled hash table buckets, but this can be easily remedied with a larger hash table or a better hash function. Ultimately, these small decreases in efficiency could not account for the overall poor scaling.

Having eliminated the other possible obstacles to scaling, we determined that the remaining cause for increasing memory stalls is the capacity of the system bus. We confirmed that the address bus reached 77% utilization on eight cores. As an engineering rule of thumb, 75% utilization is considered fully saturated for the address bus. Ultimately, *address bus saturation* was determined to be the primary obstacle to performance scaling. We maintain that if we remove this bottleneck, scaling will improve.

The remainder of our paper is presented as follows. In Section 2, we provide an overview and the background of our research problem. In Section 3, we describe our methodology for experiment and our testbed setup. In Section 4, we present our overall performance scaling results, and we systematically examine the potential bottlenecks of the key hardware and software components in Section 5. We present our conclusions and future work in Section 6.

2. BACKGROUND AND MOTIVATION

Internet client-server workloads typically involve multiple concurrent client accesses per server, where the largest servers can support thousands of clients simultaneously. The clients communicate with the server with mutually independent connections or *flows*. A multi-core server can leverage this independence to support scalability by exploiting *flow-level parallelism* [17, 10, 29].

The goals of our study were as follows:

1. to determine the performance scalability of a reference server workload on a multi-core system,

2. to expose any bottlenecks to scaling,

3. to determine if such bottlenecks are implementation-specific or fundamental to the workload, and

4. to propose solutions for any scalability problems we find.

Our performance metric is the aggregate throughput of data, packets, or connections as measured by a network application. We define *performance scaling* as the ability of a system to increase performance as hardware resources increase. Our focus is on scalability with respect to the number of cores (compute resources) on the system.

Other efforts [30] have addressed scalability of clusters of machines, while our work has focused achieving scalability on a single multi-core machine. We consider effective clusters to be necessarily built from highly scalable individual machines.

We hypothesize that the following four conditions are necessary for network application performance to scale with the number of cores:

1. flows and interrupts can be distributed and affinitized effectively to cores;

2. network applications and network protocol stacks scale well with load on a single core, and they are multi-threaded so that processing of unrelated flows incurs minimal dependencies;

3. the OS scheduler is not the bottleneck and it effectively maintains load balancing across the cores; and

4. the platform's cache and bus hierarchy are sufficiently provisioned and appropriately designed for a full workload utilizing on all cores on the system.

2.1 Flow Distribution and Affinity

Scaling on multi-core systems presents many of the same challenges as existing symmetric multiprocessor (SMP) systems. Enforcing processor affinity of interrupts and network processing has been shown as beneficial for SMP systems [17, 5], and the same benefits should apply to multi-core systems.

Distributing network processing to multiple cores can be achieved using multiple network interfaces (NICs) on a system, or affinitizing interrupts of NICs to different cores. By mapping interrupts to cores, the network protocol processing for all flows serviced by that NIC is executed to the same core. We do not expect OS schedulers to migrate processing unnecessarily, and we will confirm the performance of the scheduling policy adopted by Linux. Our experimental setup has ensured that the same core will always receive all of a flow's packets.

Receive-Side Scaling (RSS) [14] technology has evolved from these insights. RSS-enabled NICs [7] are designed to provide interrupt and flow affinity without user intervention. Supporting multiple RSS queues on a single NIC functions like multiple NICs to the system by distributing interrupts from a single physical NIC. The the hashing algorithm used by RSS maps flows to distinct cores.

2.2 Scheduling and Load Balancing

While using RSS or multiple NICs can distribute flows uniformly among cores, we must validate that this does not lead to a load imbalance in the system. Due to the unpredictability inherent when receiving remote requests, it is

not possible for an end server to accurately predict the load on a system. A request of a client can generate a response with arbitrary size and unpredictable application processing. The OS scheduler will compensate by migrating threads from busy cores to idle cores, but this can produce scheduler overheads and cache misses due to moving data to another core. Furthermore, migration of a flow's protocol processing to another core can cause the flow's packets to be processed out of order, resulting in additional overheads for protocols such as TCP.

The Linux scheduler assigns strong processor affinity to cores, but it migrates processes among cores for two reasons. The first reason is to resolve load imbalances. The other is to implement I/O affinity; when a thread blocking for I/O is signaled, it will be awakened on the core where the event occurred [16]. This ensures that application processing of of a flow's packets will likely be executed on the same core as its network protocol processing. Since a flow's network protocol processing occurs on the core determined by the NIC's interrupt, the flow's application thread will execute on the same core as well. We expect the migration of processes is kept to a minimum; the only exceptions are to ensure necessary load balancing and affinity.

Other studies [9, 19, 18] have shown load-balancing to be an issue for edge (e.g. routing) workloads. They proposed various techniques to improve load balancing while both minimizing migration of network flows and preventing packet reordering. These studies assume that the total load on a system is directly proportional the the size of a flow. Our study has validated that our workload does not cause a load imbalance and that any thread migration by the scheduler does not affect performance.

2.3 Systems and Application Software

Of particular importance is how well network protocol stacks are implemented to scale. The *de facto* standard networking protocol for many Internet workloads is TCP. The optimization of single-core TCP implementation has been addressed exhaustively over the years [4, 2]. Systems software developers continue to improve upon and optimize the TCP stack [26].

In addition to the OS, the server application must also be written to scale. We have used a web server as the reference workload. Modern web servers are multi-threaded and designed to handle large numbers of concurrent flows [12]. Due to a general lack of interaction between the network flows of different clients, the web server should represent one of the easiest types of network applications to parallelize.

2.4 Cache and Bus Hierarchy

Building upon effective flow distribution (using RSS or multiple NICs) and multi-threaded OSes and applications, a multi-core server can leverage flow-level parallelism and can potentially scale its performance with the number of cores. However, even with optimally scalable software, the design of the cache and bus hierarchy on multi-core systems can influence the scalability of Internet server workloads [27, 8, 3]. Sharing cache between cores, for instance, may reduce data communication overhead between the cores, but it may also cause one core to evict the other core's data. Multiple caches may incur overhead if they contend for the same bus. Our analysis has validated whether system caches and buses are adequately provisioned.

We present our analysis of each of these scalability conditions in Section 5, but first, we must describe our experiment methodology and testbed setup in Section 3 and our overall performance scaling results in Section 4.

3. EXPERIMENT METHODOLOGY

To determine how well a typical Internet server workload scales on a multi-core system, we chose a web server as our workload on an eight-core system. Since clients interact with web servers via independent connections, they provide the opportunity to exploit flow-level parallelism on a multi-core server.

3.1 SPECweb2005

We chose the SPECweb2005 benchmark to be our reference web workload [20, 6]. SPECweb2005 is a suite of synthetic workloads that emulates large numbers of independent web clients and provides files and scripts for producing web content.

Of the three workloads SPECweb2005 provides, we chose to exclusively use the Support workload. Unlike the Banking and E-Commerce workloads of this benchmark suite, this workload has no encrypted connections and emulates file downloads from Internet clients. We chose the Support workload because it provided the simplest possible abstraction of how a realistic application interacts with network processing. It also produced the largest networking load, so it was best suited to understand the scalability of the network stack under heavy load.

3.2 Network Protocol Stack Isolation

In addition to a web server, to isolate scaling bottlenecks specific to network processing, we conducted experiments using a workload with a trivial application. The SPECweb2005 Support workload primarily uses the bulk-data, zero-copy transmit and receive-side bandwidth makes up about 1/50th of transmit bandwidth [6]. We used Netperf to emulate this workload by loading up to 8Gb/s of zero-copy full-packet transmits; and up to 800Mb/s (rate throttled) worth of full-packet receives.

3.3 SPECweb2005 Modifications

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 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 50 disk drives for the file set [21], 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.

Table 1: Server Kernel Settings

Setting	Value
fs.file-max	5000000
net.core.netdev_max_backlog	400000
net.core.optmem_max	10000000
net.core.rmem_default	10000000
net.core.rmem_max	10000000
net.core.somaxconn	100000
net.core.wmem_default	10000000
net.core.wmem_max	10000000
net.ipv4.conf.all.rp_filter	1
net.ipv4.conf.default.rp_filter	1
net.ipv4.tcp_congestion_control	bic
net.ipv4.tcp_ecn	0
net.ipv4.tcp_max_syn_backlog	12000
net.ipv4.tcp_max_tw_buckets	2000000
net.ipv4.tcp_mem	30000000 30000000 30000000
net.ipv4.tcp_rmem	30000000 30000000 30000000
net.ipv4.tcp_sack	1
net.ipv4.tcp_syncookies	0
net.ipv4.tcp_timestamps	1
net.ipv4.tcp_wmem	30000000 30000000 30000000

Table 2: Web Server Threads Settings

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

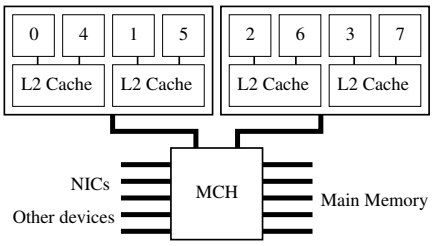

Figure 1: System design of the test server.

Changing this option affects the system in two ways. First, it removes disk I/O traffic from the system bus and memory. This means that performance using disks could be worse than our results. Secondly, it increases temporal locality of reference, since it reduces the total number of files This does not improve miss rates in the CPU cache, since we configured Apache to use the *sendfile* system call. Thus, files contents are sent to the NICs via DMA without entering the CPU cache.

To perform large number of experiments more quickly, we reduced the *ITERATIONS* option from 3 to 1. SPEC requires three consecutive successful runs for an official valid result. We also reduced the *RUN_SECONDS* option to 300 from the normally required 1800 seconds. To improve the stability and consistency of the results, we increased the *MAX_OVERTHINK_TIME* option from 20000 to 2400000 milliseconds. This option sets the time a client thread is allowed to exceed its "think time" between requests. For unknown reasons, some client threads became stalled during some runs. After increasing the option, there was no discernible performance difference reported in the results.

3.4 Server Software

The test server ran the Gentoo 1.12.9 operating system with Linux kernel 2.6.20.3, which was compiled with support for the Intel 64 (x86_64) architecture. We chose software with freely-available source code to ease profiling. We used Apache HTTP Server 2.2.4—a multi-threaded server that provides one thread per flow and scales at least to tens of thousands of concurrent sessions [12]. We also used the PHP 5.2.1 script engine for the SPECweb2005 PHP scripts, and we used the eAccelerator 0.9.5 bytecode caching engine which improves the performance of PHP scripts.

Table 1 shows modifications made to the Linux kernel default *sysctl* settings [11]. We increased operating system limits where necessary to scale. We also set TCP options to values we believe are similar to those used on high-performance commercial web servers.

We also configured the Apache HTTP Server to improve scalability. We used the *mpm_worker_module* which provides a combined multi-process and multi-thread server with one thread per connection. Table 2 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 user space.

3.5 Server Configuration

The web server contained an Intel S5000PSL Server Board, two Intel Xeon X5355 processors, and 16GB of main memory. Figure 1 shows the arrangement of the processors, L2 cache, system buses, and the memory controller hub (MCH). The processors have four cores each (numbered 0–7) with an L2 cache per pair of cores. Each processor socket is connected by its own front-side bus to the MCH. The MCH is connected to the memory, NICs, an I/O controller hub, and other devices.

The cores in the figure are numbered in the (BIOS-specific) order that the operating system saw them. Thus, when configured with less than the maximum number of cores, the OS enabled the cores in order. For example, booting the system with four cores would enable only cores 0–3.

3.6 Testbed Configuration

The testbed consists of four client machines and the server (system under test). For our microbenchmark tests, we used eight 1GbE (Intel PRO/1000) NICs with each NIC mapped to one of eight cores. For our SPECweb2005 tests, each of the four client machines is connected to one of the four 1GbE NICs on the server. Interrupts on each NIC are affinitized to cores 0–3. This ensures that the processing and interrupts of the flows on each NIC are directed to a distinct core.

SPECweb2005 also requires at least one backend database simulation server as part of the testbed. For simplicity, we configured each client machine to double as a backend server.

3.7 Measuring Performance

We gathered results of our experiments from the outputs of SPECweb2005 and OProfile system analyzer. For each experiment (except for the single-core experiments) we pushed

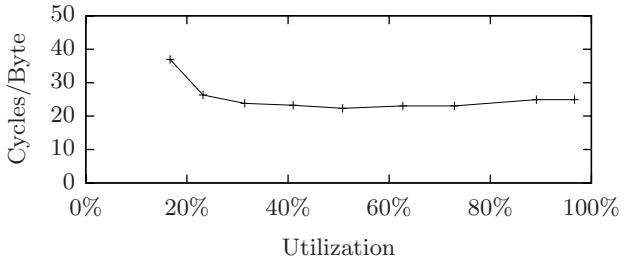

Figure 2: Efficiency of a single-core web server workload under a range of loads.

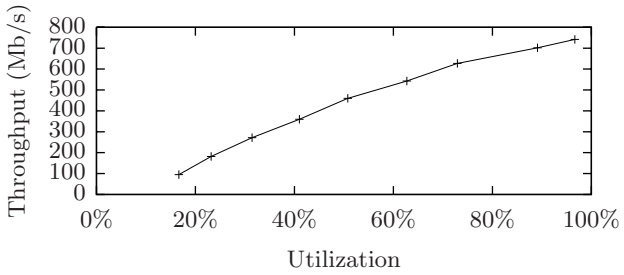

Figure 3: Transmit throughput of a single-core web server workload under a range of loads.

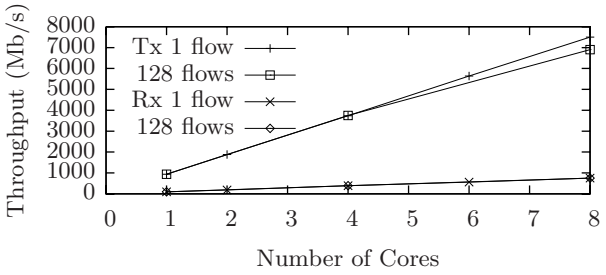

Figure 4: Throughput scaling of network processing with cores (1 and 128 flows per core).

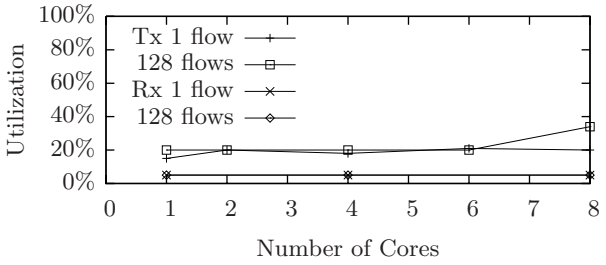

Figure 5: Utilization scaling of network processing with cores (1 and 128 flows per core).

the server to the highest number of concurrent sessions (in steps of ten) which results in an acceptable response time as defined by SPECweb2005: at least 95% "good" requests, at least 99% "tolerable" requests.

From these runs we use the SPECweb2005 results output to determine the total number of data bytes transmitted by the server and thus derive the throughput by dividing by the 300-second run time.

We ran OProfile for about 100 seconds of the 300-second run, and used it to report the approximate number of cycles, instructions executed, L2 cache misses, DTLB misses, data bus transactions, and address bus transactions, both overall an per function. Since the server's processors support only one profiler event at a time, we performed a separate benchmark run for each event. To compute utilization and remove idle time from our results, we subtracted the number of OProfile events the kernel spent in the *poll_idle*, *enter_idle*, *exit_idle*, *__exit_idle*, and *cpu_idle* functions.

To report scalability in terms of the workload, we use the number of utilized events per byte transmitted. By *utilized events*, we mean the number events in the non-idle functions. By *bytes transmitted*, we mean the number of packet payload bytes received by the SPECweb2005 clients.

Now that we have described how we conducted experiments and measured results, we can present our performance scaling results of our server in the next section, and then we shall analyze scaling problems in Section 5.

4. PERFORMANCE SCALING RESULTS

Before presenting our results for a multi-core workload, we first show that our workload behaved as expected on a single core without multi-core interactions and dependencies. We then look at the performance scalability of kernel's network protocol stack in isolation as a comparison point before moving on to present the scalability results of the full web server workload.

4.1 Single-Core Performance Baseline

Figure 2 shows how the web server performed on a single core as we throttled the load to vary CPU utilization. The efficiency of processing (as measured by cycles per byte) remained constant up to 96.7% utilization. Additionally, Figure 3 confirms that throughput scaled nearly linearly with load. We have therefore established that the web server scales well with load on a single core, and we can proceed to further analysis with a valid single-core baseline.

4.2 Network Protocol Stack Scalability

We isolated the performance scalability of the network protocol stack from that of the web server application using the methods discussed in Section 3.2. Figures 4 and 5 show that network protocol processing scaled linearly with the number of cores (one flow per core), as well as with the number of flows (128 flows per core). Our goal was 8Gb/s for transmit and 800Mb/s for receive on eight cores. Since Netperf creates one process per flow, the load of 128 flows per core resulted in a total 1024 processes concurrently sending and receiving on the system.

We loaded the system with more throughput and process scheduling than we expect from the full web server workload runs (in the rest of this study). These results have validated that, if flows and interrupts are affinitized evenly (with minimal application overhead) and the Linux network stack and the drivers are implemented as expected, then throughput scales with the number of cores.

4.3 Web Server Scalability

We experimentally determined the performance scaling of a web server workload using the setup described in Section 3.

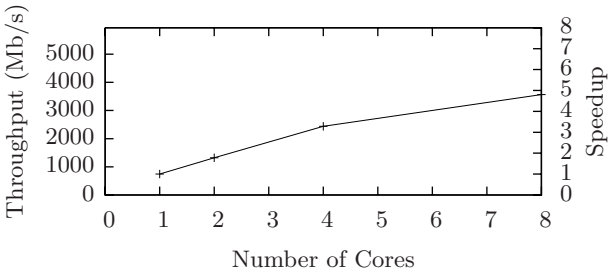

Figure 6: Web server transmit throughput scalability with the number of cores. Relative speedup is shown on the right.

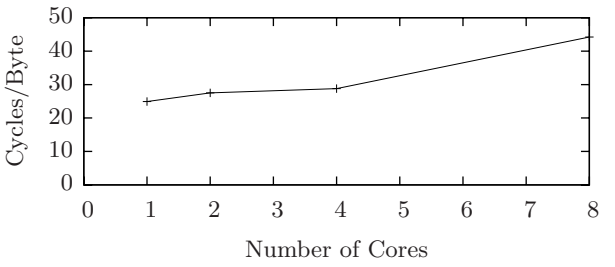

Figure 7: Cycles per transmitted byte as the number of cores increases.

Figures 6 and 7 show how well the web server scales as the number of cores increases. Throughput is computed from the benchmark results, and it represents the rate of data transmitted by the web server minus the packet headers. The term *cycles/byte* refers to the total number of cycles taken by the server to transmit each byte of data. This normalized metric is key to our assessment of how efficient processing is as core count varies.

As shown in Figure 6, the throughput drops far below a linear speedup on eight cores. Figure 7 shows an 77% increase in cycles per byte from one to eight cores. There is a 54% increase in cycles per byte from four to eight cores alone.

In the next section, we systematically analyze the potential causes of the web server's poor performance scaling.

5. PERFORMANCE SCALING ANALYSIS

In Section 2, we posed four requirements for network application performance scaling. To determine why the web server scales poorly, we present our analysis of each requirement in turn.

5.1 Flow Distribution and Affinity

For performance to scale, flows should be distributed to multiple cores to prevent bottlenecks, and packets within a flow are distributed to the same core to prevent packet reordering and data sharing between cores. To perform experiments in this paper, we used multiple NICS and we assigned interrupts to cores to maintain exact control on how flows are directed. Thus, by design, our results do not reflect problems with the distribution and affinity of flows.

However, we envision that commercial servers will leverage RSS, and must therefore establish that RSS-enabled NICs can effectively distribute flows uniformly and independently across cores. RSS computes the destination core for a flow by computing the Toeplitz hash function on the 5-tuple that uniquely identifies a TCP flow. It then uses the result to look up the core in a table [14].

Table 3: Toeplitz Hash Randomness Test Results

	Banking	E-Commerce	Support
number of flows	18170	13191	10569
entropy	7.99	7.99	7.98
percent error	0.12%	0.17%	0.23%
χ^2	50%	50%	50%
arithmetic mean	127.5	127.6	126.7
percent error	0.04%	0.08%	0.66%
Monte Carlo π	3.17	3.16	3.11
percent error	1.04%	0.62%	1.16%
serial correlation	−0.0071	−0.0122	0.0000

One method of testing the uniformity and independence of a hash function result is to test the function's efficacy as a pseudo-random number generator. We used the Ent [28] program perform five randomness tests on the sequence of values produced by the Toeplitz hash function. These values were computed on TCP 5-tuples found in traces captured from benchmark runs for each of the three SPECweb2005 workload types. These tests include computing the entropy, performing a Chi-square (χ^2) test, computing arithmetic mean, computing the Monte Carlo value for π, and computing the serial correlation coefficient. Although RSS uses seven bits for the table lookup, Ent works with 8-bit values, so we used the lower eight bits of the Toeplitz hash function result to perform the randomness test.

Table 3 presents the randomness test results. The χ^2 test cannot discern between the sequence of hash values and a random sequence. The other four tests confirms that the hash values are distributed uniformly and independently with a very low percent error as compared to the ideal "perfectly random" result. These results confirmed that the Toeplitz hash function produces uniform and independent results.

5.2 Scheduling and Load Balancing

While we have established that flows are distributed uniformly and independently, these flows may put different levels of load on the system depending upon the type of request. One or more cores may become fully loaded before others, such that the idle time on other cores result in time wasted on non-useful work. The purpose of load balancing is to fully utilize the system by scheduling busy application threads on idle cores, but migrations can result in costly cold caches and TLB misses on the destination core.

We designed an experiment to isolate the effects of migration and load imbalance by *removing migrations completely*. We compared our standard setup with a "separated" setup that runs four independent instances of Apache with threads hard-affinitized to each of four cores with unshared L2 cache. Interrupts for each of the four NICs were affinitized to each of the four cores. Thus, each web server instance has its own core, NIC, and network protocol processing independently from the rest. Table 4 compares the result of running the modified SPECweb2005 workload with combined and separated instances.

Table 4 shows that the throughput, utilization, cache miss

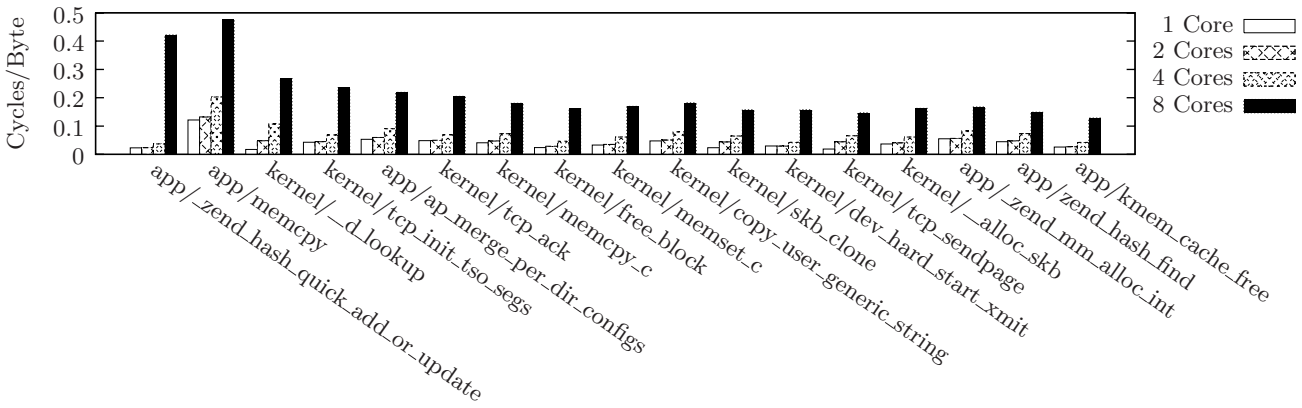

Figure 8: Functions with the largest cycles/byte increase from one to eight cores.

Table 4: Performance of Standard and Separate Application Instances

	standard	separated
cores	4	4
throughput (Mb/s)	2437	2494
utilization	98.0%	96.6%
L2 misses/byte	0.0638	0.0623
cycles/byte	28.8	29.6

rates, and cycles/byte are similar between the standard and separate-instance cases, with a slight advantage to running with separate instances. Given the high utilization in the separate-instance case, we can conclude that there is no load imbalance inherent to the SPECweb2005 Support workload. From the similar performance in the standard case, we can conclude that the OS scheduler does not significantly create imbalance or add thread migration overhead to the already-balanced workload. If we assume that SPECweb2005 accurately models realistic web workloads, we can conclude that web servers do not face a load imbalance problem. We further confirmed that the OS scheduler is effective at handling such workloads, since it did not introduce imbalances or migrations. Ultimately, we confirmed that load imbalance does not contribute to poor scalability of web servers.

5.3 Systems and Application Software

Aside from migration and load balancing, poor scalability of software in a shared memory environment can also be caused by the following problems:

1. lock contention for data sharing,
2. iteration through large, shared data structures,
3. contention for shared atomic variables, and
4. cache and TLB misses from core-to-core data migration.

These problems could arise in either the operating system kernel or the application. Figure 9 shows the utilization of both the kernel and application as the number of cores was changed. Even as the overall efficiency decreased with the number of cores, the ratio of kernel-to-application utilization remained nearly one-to-one. Thus, scaling bottlenecks affected both the application and kernel equally.

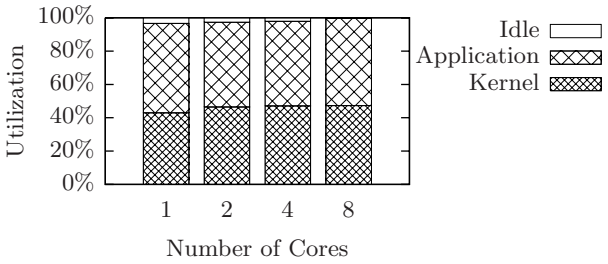

Figure 9: Utilization of the kernel and application as the number of cores increases.

We profiled functions for both the kernel and application on the source code and assembly levels to determine which of them contributed the most to poor performance scaling. Figure 8 shows the functions with the largest overall increase in cycles per byte from one to eight cores. Together, they account for a third of the total increase in cycles per byte. The number of cycles per byte for each of these functions more than doubles from four to eight cores.

Of the twelve kernel functions listed, nine were memory intensive. They included instructions for allocating, deallocating, copying, or initializing memory that often resulted in cache misses. These instructions produced longer stalls as the number of cores increased. Two functions, *tcp_sendpage* and *skb_clone*, increased in the time spent in atomic operations associated with maintaining multiple references to socket buffer data structures. The remaining function, *__d_lookup*, was primarily affected by contention for spin locks.

Of the five functions in application space, memory instructions stalls, like those found in the kernel, appeared again in *memcpy* and *_zend_mm_alloc_int*. The remaining functions, *_zend_hash_quick_add_or_update*, *ap_merge_per_dir_configs*, and *zend_hash_find* traversed linked lists that appeared to increase in size as the workload increased.

Next we examine each of these functions for the aforementioned programming problems which may adversely affect performance scaling in a shared-memory environment.

5.3.1 Lock Contention for Data Sharing

The least scalable kernel function in Figure 8 was *__d_lookup*, which is part of the filesystem. While network flows were

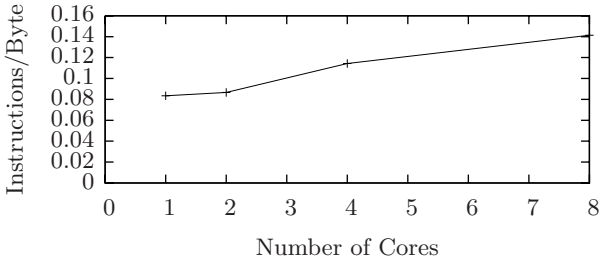

Figure 10: The number of instructions executed per byte transmitted for the kernel function *__d_lookup*.

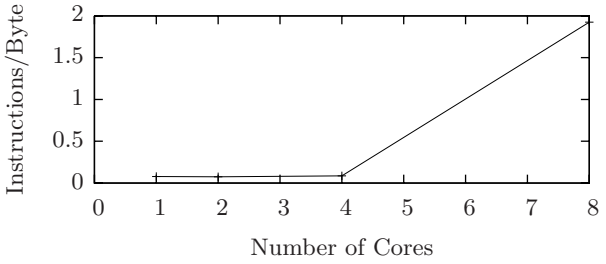

Figure 11: Instructions per transmitted byte for PHP function *_zend_hash_quick_add_or_update*.

independent, the filesystem is a shared resource separate from the network protocol stack. Therefore, the benefits of flow-level parallelism do not apply.

Each time an application opens a file, the kernel translates its path name into a filesystem node reference. The kernel walks through each directory in the path to find a reference to the next file or directory in its directory cache. It calls *__d_lookup* repeatedly to scan all the entries in each directory for a match. The function acquires a spin lock for each directory entry it scans. In our experiments, the web server application opened several files for each web request. As the number of concurrent web requests increased, contention for spin locks also increased.

If spin lock contention has reduced our performance, then it must have lengthened the code path resulting in a larger number of instructions executed. Figure 10 shows the number of instructions executed per byte of data transmitted for the *__d_lookup* function. As the number of cores increased, the instruction count also increased due to spinning to acquire the lock.

5.3.2 Shared Data Structures

We used the same measure of instructions per byte to determine whether the number of iterations needed to traverse large data structures have increased with the workload. Figure 11 isolates the instruction count per byte transmitted for the function *_zend_hash_quick_add_or_update*. Analysis of the function's profile shows that it spent more time walking linked lists in hash table buckets, suggesting that a hash table contained too many entries in the eight-core run. Simply increasing the table's size may be sufficient to improve performance. Profiles of other poorly-scaling application functions such as *ap_merge_per_dir_configs* and *zend_mm_alloc_int* also reveal growing numbers of instructions per byte in linked list walks.

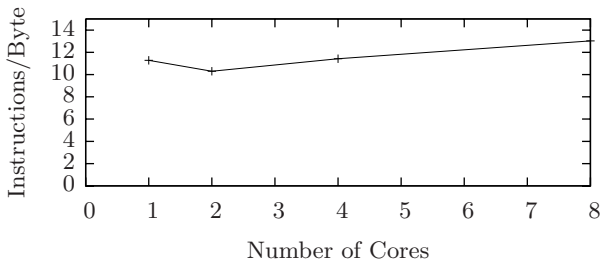

Figure 12: System-wide number of instructions executed per byte transmitted as the number of cores varies.

As shown in Figure 12, the overall number of instructions executed per byte increased only slightly from one to eight cores. The function *_zend_hash_quick_add_or_update* accounts for nearly all of the increase shown in the figure. This reveals that the total increase in instruction count is mostly accounted for with just the functions discussed in this section. While these functions stand out, the overall increase in code path was minimal.

5.3.3 Contention for Shared Atomic Variables

Profiling revealed that the kernel networking functions *tcp_sendpage* and *skb_clone* use atomic operations which scaled poorly as the number of cores increased. The function *tcp_sendpage* implements zero-copy transmit for TCP, which was used extensively by the Apache web server to transmit files. The other function, *skb_clone*, creates "fast cloned socket buffers" where one clone is transmitted while the other is kept for possible retransmission. Both functions perform atomic operations on the clones' reference counts [15].

While it is possible that this atomic reference count variable was accessed on multiple cores, contention between cores for the variable is unlikely due to flow affinity. Creation of this socket data structure occurs on the application thread's core as a result of the *send* system call, which may or may not share the same core as the NIC's interrupt. However, all subsequent references to the atomic variable occur on the same core as the NIC's interrupt. Deallocation of the transmitted clone's memory is triggered by the NIC's interrupt and occurs on the interrupted core. The deallocation of the backup clone's memory also occurs on the same interrupted core when a TCP acknowledgment arrives. A rare exception is an actual retransmission, which can occur in the context of the TCP retransmission timer.

As a result, while there is no inherent reason for contention for the atomic reference counts between cores, we still observed reduced performance of these atomic operations as the number of cores increased. The root cause of this unexpected behavior will be addressed when we discuss the system address bus in Section 5.4.2.

5.3.4 Core-to-Core Data Migration

We expected that, due to flow-level parallelism, different cores have little reason to share data. If data sharing has occurred, it should be reflected in in cache and TLB misses on the destination cores. A large number of kernel and application functions which which exhibit poor scalability due to memory load and store stalls, which may indicate cache

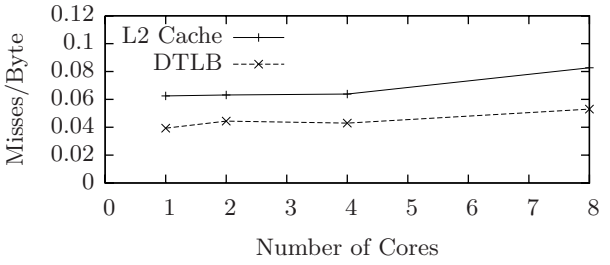

Figure 13: L2 Cache and DTLB misses per instruction.

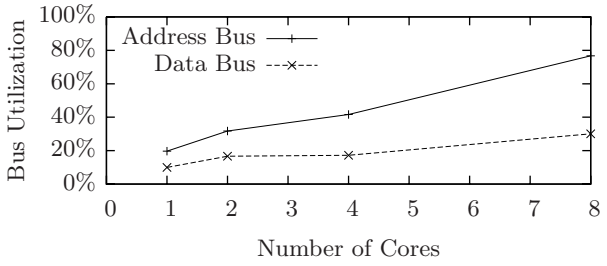

Figure 14: System data and address bus utilization.

thrashing. These functions incur large memory load and store stalls for memory that is not likely to be in cache, such as initializing newly allocated memory or updating "previous" and "next" pointers in the memory lists which have become cold in cache. Figure 13 validates that the overall number of L2 cache and DTLB misses per byte transmitted varies little as the number of cores increases.

There was however, an increase in misses per byte when going from four to eight cores. Since we have established that the workload behaved as expected between one and four cores, the increase when going to eight cores stands out. Only when the workload runs on more than four cores do pairs of cores share L2 cache (see Figure 1, Section 3.5), suggesting that the contention between cores is responsible for more frequent cache evictions. Experimental results in the next section will show that cache sharing had little effect on performance since the effect of increased misses per byte was minimal.

As with the case of atomic variables, actual contention for data contributed little to increased delay to handle memory stalls. This behavior will be examined in depth in Section 5.4.2.

5.4 Cache and Bus Hierarchy

Section 5.3 revealed a atomic operation and memory stalls that could not be explained by contention between cores for shared data. Aside from the workload and software, system design decisions such as shared caches, shared buses, and bus capacity can also affect these delays.

5.4.1 Shared Cache

We wanted to determine if the existence of a shared L2 cache affected the performance of our workload. On one hand, cache sharing provides a faster means of sharing data between the cores than using the front-side bus. On the other hand, code running on each core may evict each other's data from cache due to capacity misses.

We set up an experiment comparing the modified SPECweb2005 Support workload running on two cores with a shared L2 cache versus two cores with separate L2 caches. For these experiments, the other cores on the system were on-line but were not used by the web server or the NICs. Using the numbering scheme shown in Figure 1, Section 3.5, cores 0 and 4 were used in the shared-cache run, while cores 0 and 1 were used in the unshared run. The results showed that shared cache produced a throughput of 1323Mb/s, while unshared cache produced a close 1305Mb/s. Thus, we have validated that sharing L2 cache had little effect on overall performance for our workload.

5.4.2 Bus Utilization

Snoopy cache coherence and contention for shared system buses may also adversely affect performance scaling. Figure 14 shows the utilization of the address and data buses on the test server as the number of cores increases. The data bus is a set of system bus lines which carry cache line data, while the address bus is a separate set of bus lines which carry the physical addresses for all cache-to-cache and cache-to-memory snoop requests and responses. The data bus utilization increases from one to two cores and from four to eight cores, but not from two to four cores. The address bus utilization, however, always increases, including from two to four cores.

The system design of the test server explains this behavior. Figure 1 in Section 3.5 shows that there was one bus for each CPU socket. When going from two to four cores, the server went from using one socket to two and thus doubled its bus capacity. Data bus traffic on one bus did not affect the other bus. However, address bus traffic was propagated between the two buses due to cache snoops.

As we determined in Section 5.3, increased memory stall delays represent the primary cause of poor scalability. We validated that cache, TLB miss rates, and the number instructions executed did not increase significantly per unit of work as the amount of workload increased. This definitively shows that saturation of the address bus is the only remaining explanation for the longer memory stall delays, and accordingly, it represents the key bottleneck to performance scaling.

6. CONCLUSIONS

Our results have shown that, due to flow-level parallelism in web server workloads, the number of cache and TLB misses remained nearly constant per byte as the number of cores increased. Likewise, shared cache between cores on the same bus had little effect on performance when compared with unshared cache. Because of flow-level parallelism, there was little data shared between caches.

Profiling revealed some scalability obstacles in software. On the application side, a larger workload resulted in overfilled data structures. However, increasing hash table capacities and reducing dependence on linked lists that grow with the workload should remedy these scalability problems. Most importantly, the application spent little time locking and sharing data between cores since it took advantage of flow-level parallelism by assigning one thread per connection.

In the kernel, flow-level parallelism broke down in the file-system directory cache which was shared system-wide. The

kernel's directory lookup spends increasing amounts of time contending for spin locks as the workload increased. Since the contention is per-directory entry, a possible workaround would be to maintain alternate directory trees for each core.

For the networking portion of its workload, the kernel implemented flow-level parallelism well. We have shown that, without the influence of the application and filesystem, the network stack scaled well with the number of cores for one to many flows.

For both the kernel and the application, we showed that memory stall delays increased. We determined that the address bus utilization increased and became saturated on eight cores. We conclude that the address bus capacity is the primary obstacle to performance scaling for network applications.

Developments such as Intel QuickPath and HyperTransport can reduce contention by replacing the shared bus and central memory controller with point-to-point links. However, since broadcasting snoops is expected to grow $O(n^2)$ with the number of cores, the number of links would have to grow on the same order to scale. Directories (and directory caches) can be used to replace snoopy cache coherence, but this requires additional cost and latency [13].

Our future efforts will focus on addressing this bottleneck. We want to explore more accurate models that relate bus capacities to core count for realistic workloads. We believe that improved bus filtering and partitioning techniques should be explored. When such issues are addressed, commercial web server and systems software will be well-positioned to scale to a large number of cores.

7. ACKNOWLEDGMENTS

The authors would like to thank Vish Viswanathan, Tong Li, and Wayne Swick for their support.

8. REFERENCES

[1] CAIDA. *Workload Characterization: Application Cross-Section*, 2006. http://www.caida.org/analysis/workload/.

[2] J. Chase, G. Gallatin, and K. Yocum. End-system optimizations for high-speed TCP. 39(4), 2001.

[3] S. Chinthamani and R. Iyer. Design and evaluation of snoop filters for web servers. In *SPECTS*, 2004.

[4] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), 1989.

[5] A. Foong, J. Fung, D. Newell, A. Lopez-Estrada, S. Abraham, and P. Irelan. Architectural characterization of processor affinity in network processing. In *ISPASS*. IEEE, 2005.

[6] R. Hariharan and N. Sun. Workload characterization of SPECweb2005. In *SPEC Benchmark Workshop*. SPEC, 2006.

[7] Intel Corporation. *Receive Side Scaling on Intel Network Adapters*. http://support.intel.com/support/network/adapter/pro100/sb/CS-027574.htm.

[8] R. Iyer. Characterization and evaluation of cache hierarchies for web servers. *World Wide Web*, 7(3):259–280, Sept. 2004.

[9] L. Kencl and J.-Y. L. Boudec. Adaptive load sharing for network processors. In *INFOCOM*, volume 2, pages 545–554. IEEE, 2002.

[10] É. Lemoine, C. Pham, and L. Lefèvre. Packet classification in the NIC for improved SMP-based Internet servers. In *ICN*. IEEE, 2004.

[11] Linux Kernel. *Linux IP Sysctl Documentation*. Documentation/networking/ip-sysctl.txt.

[12] C. MacCárthaigh. Scaling Apache 2.x beyond 20,000 concurrent downloads. In *ApacheCon EU*, July 2005.

[13] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *ISCA*, pages 206–217, 2003.

[14] Microsoft Corporation. *Scalable Networking with RSS*, Apr. 2005.

[15] D. Miller. *How the Linux TCP Output Engine Works*. http://vger.kernel.org/~davem/tcp_output.html.

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

[17] J. Salehi, J. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version). *Transactions on Networking*, 4(4):516–530, Aug. 1996.

[18] W. Shi and L. Kencl. Sequence-preserving adaptive load balancers. In *ANCS*, pages 143–152. IEEE/ACM, Dec. 2006.

[19] W. Shi, M. MacGregor, and P. Gburzynski. Load balancing for parallel forwarding. *Transactions on Networking*, 13(4):790–801, Aug. 2005.

[20] SPEC. *SPECweb2005 Release 1.10 Benchmark Design Document*, Apr. 2006.

[21] SPEC. *SPECweb2005 Result for the HP ProLiant DL380 G5*, 2007. http://www.spec.org/web2005/results/res2007q2/web2005-20070507-00066.html.

[22] SPEC. *SPECweb2005 Result for the HP ProLiant DL385 G2*, 2007. http://www.spec.org/web2005/results/res2007q3/web2005-20070828-00079.html.

[23] SPEC. *SPECweb2005 Result for the HP ProLiant DL580 G5*, 2007. http://www.spec.org/web2005/results/res2007q3/web2005-20070828-00077.html.

[24] SPEC. *SPECweb2005 Result for the HP ProLiant DL585 G2*, 2007. http://www.spec.org/web2005/results/res2007q2/web2005-20070507-00067.html.

[25] SPEC. *SPECweb2005 Result for the HP ProLiant ML360 G5*, 2007. http://www.spec.org/web2005/results/res2007q2/web2005-20070507-00068.html.

[26] S. Tripathi. FireEngine—a new networking architecture for the Solaris operating system. Whitepaper, Sun Microsystems, Nov. 2004.

[27] V. Viswanathan. Intel front side bus architecture. Intel Software College course, 2006.

[28] J. Walker. *Pseudorandom Number Sequence Test Program*. Fourmilab, Oct. 1998.

[29] P. Willman, S. Rixner, and A. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *USENIX*, pages 91–96, 2006.

[30] W. Zhang and W. Zhang. Linux virtual server clusters. *Linux Magazine*, 5(11), 2003.