Comparing High-Performance Multi-core Web-Server Architectures

Ashif S. Harji Peter A. Buhr Tim Brecht

School of Computer Science
University of Waterloo
asharji,pabuhr,brecht@uwaterloo.ca

Abstract

In this paper, we study how web-server architecture and implementation affect performance when trying to obtain high throughput on a 4-core system servicing static content. We focus on static content as a growing numbers of servers are dedicated to workloads comprised of songs, photos, software, and videos chunked for HTTP downloads. Two representative static-content workloads are used: one serviced entirely from the file-system cache and the other requires significant disk I/O. We focus on 4-core systems as: 1) it is a widely used configurations in data-centers and cloud services, 2) recent studies show large SMP systems may operate more efficiently when subdivided into smaller subsystems, 3) understanding performance with a smaller number of cores is essential before scaling to a larger number of cores, 4) and 4-cores may be sufficient for many web servers.

Two high-performance web-servers, with event-driven (μserver) and pipelined (WatPipe) architectures, are developed and tested for a multi-core environment. By carefully implementing and tuning the two web-servers, both achieve performance comparable to running independent copies of the server on each processor (N-copy). The new web-servers achieve high throughput (4,000–6,000 Mbps) with 40,000 to 70,000 connects/second; performance in all cases is better than nginx, lighttpd, and Apache. We conclude that implementation and tuning of web servers is perhaps more important than server architecture. We also find it is better to use blocking rather than non-blocking calls to sendfile, when the requested files do not all fit in the file-system cache.

1. Introduction

One of the biggest problems for many Internet companies is handling huge volumes of traffic resulting from a large number of users. With social networking and cloud computing growing in popularity, not only is more user generated and commercial content moving online, ever larger user-communities are placing increasing demands on servers while accessing this content. For example, Facebook serves over 1,000,000 images per second at peak [5]. Delivering this content *economically* is a growing concern. The *web-server* is the software component through which most of this Internet traffic flows, hence it must provide high throughput while supporting a large number of concurrent connections. A major challenge in implementing high-performance web-servers is exploiting the processing power available on multi-core servers to *efficiently* handle these loads, which is increasingly important and is not a solved problem.

To achieve high performance, many sites use highly-tuned web-servers for various types of traffic. Static content remains an important and significant component of web traffic, e.g., software repositories, songs, photos, and videos chunked for HTTP downloads [6]. Hence, specialized servers exist to handle static content [16, 18], possibly off-site as with Amazon S3 or Akamai. Even for general web-servers, efficiently handling static content frees up resources for other work or types of traffic, e.g., where several web sites or virtual machines are hosted on the same machine [30]. We believe it is crucial to first understand efficient servicing of static workloads on a small number of cores before attempting to scale to many cores or attacking more complex dynamic workloads.

This paper examines how web-server architecture and implementation affect performance when trying to obtain high throughput with a large number of connections on a 4-core system servicing static content across two representative workloads: one serviced entirely from the file-system cache and the other requiring significant disk I/O. In contrast, our previous work [24] only covered *one* workload on a *single-core* server. Interestingly, the 4–6 core CPU is one of the most prevalent and widely used configurations available in data-centers and cloud services [12, 19]. In fact, recent studies suggest large multi-core systems may operate more efficiently when subdivided into smaller subsystems [27]. This work shows 4 cores are sufficient to achieve extremely high throughput (4,000–6,000 Mbps), well beyond what most web-sites need from a single server (nearly 62 TB would

SYSTOR'12, June 4–6, 2012, Haifa, Israel.

be transferred over 24 hours at 6,000 Mbps). The goal is to demonstrate how web-server design and implementation can achieve this high throughput on a restricted multi-core platform, and to identify the underlying factors contributing to this performance. The demonstrated improvements in web-server efficiency can reduce the size and/or number of servers needed by commercial sites, cutting procurement, power, space and maintenance costs. However, optimizing parallelism requires advanced web-server design and implementation. The transition from single-core to multi-core introduces complexity both in the web-server and the operating system, e.g., CPU affinities (IRQ/process), memory footprint, shared processing, and shared memory/contention. We achieved high performance by using CPU/IRQ affinities to partition the hardware/web-server, and for workloads requiring disk I/O reduce the memory footprint by sharing resources, such as the application cache-table, using blocking sendfile, and fixing a file-system caching problem in the Linux kernel.

The contributions of this work are: 1) Two high-performance web-servers, with event-driven (μserver) and pipelined (WatPipe) architectures, are extended and tested for shared-memory multi-core environments; their performance is analysed and shown to be better than Apache, lighttpd, and Nginx. 2) Both in-memory and disk I/O workloads are examined, whereas prior multiprocessor work has focused on only in-memory. 3) In contrast to prior work [34, p. 247], we demonstrate shared-memory web-servers offer comparable performance with their non-shared-memory N-copy counterparts across both workloads. 4) In contrast to previous work [23, 24, 31, 33, 34], we demonstrate server architecture has only a small effect on performance provided a server is well-designed, implemented, and tuned. 5) We demonstrate that server throughput is higher with non-blocking sendfile when requests are serviced from the file-system cache, while the use of blocking sendfile provides better performance when requests require disk I/O. 6) A patch was developed for a serious file-system caching problem discussed in [15, § 2.3], which fixes a problem causing poor disk performance.

2. Background and Related Work

Web-server architectures are often classified by how they handle potentially-blocking network-I/O. Combining non-blocking network-I/O (socket) operations with an event mechanism like select or poll allows a server to interleave processing of tens of thousands of simultaneous connections. Several *event-driven server* architecture variants exist, e.g., Single-Process Event-Driven (*SPED*) architecture [23].

On a multiprocessor, a natural extension to SPED is to run a single-copy of a SPED server on each processor, called an *N-copy server* [34]. To deal with blocking I/O, a further extension is to run multiple server-copies per CPU; when one process blocks on I/O, another process is available to run. However, each N-copy process listens for connections on a different TCP port, thus requiring additional processing to balance requests across the N copies. To eliminate explicitly balancing requests, the concurrent SPED processes can share a listening socket, called a Symmetric Multi-Process Event-Driven (*SYMPED*) server [24]. Another extension to SPED, to deal with blocking disk-I/O, is the Asymmetric Multi-Process Event-Driven (*AMPED*) architecture [23], where helper threads handle blocking system-calls for disk I/O so the main event-driven process does not block.

Another method for dealing with blocking I/O is to use threads, called a *thread-per-connection server* (or *threaded server*); if a thread blocks, other unblocked threads can execute. Each thread handles a single HTTP request before processing another request. Apache [2] (kernel threads) and Knot [31] (user and kernel threads) use this approach.

Without kernel support for asynchronous I/O, servers must employ some form of threading to mitigate the effects of blocking disk-I/O. (In Linux, asynchronous I/O mechanisms exist but not for performance-critical system-calls like sendfile.) Furthermore, on multi-core systems, there is evidence that server architectures must incorporate both events and threads to maximize performance [31, 33].

The pipelined (or hybrid) architecture uses events and threads, but the number of threads is far fewer than the number of connections. In this approach, execution is broken into separate stages, with thread pools to service each stage. The Staged Event-Driven Architecture (*SEDA*) [33] is a complex pipelined architecture used to construct the Haboob web-server. SEDA extends the pipelined design with a dynamic resource-controller to adjust thread allocation, scheduling, and admission control at each stage to meet targets.

Previous multi-core web-server studies show significant performance benefits by pinning NIC interrupt handlers to different processors and scheduling the web-server process handling requests from these NICs on the same processor [1, 8, 13]. When this occurs, the process and interrupts affinities are aligned or *partitioned*. Specifically, partitioning means the threads processing a request execute on the same CPU handling the network interrupts for the subnet associated with the request. Partitioning benefits come from improvements in cache misses, pipeline flushes and locking.

Zeldovich *et al.* [34] implement libasync-smp to simplify writing event-driven applications for multiprocessors. They compare a libasync-smp web-server with Flash (AMPED), Apache (thread-per-connection) and N copies of a single-process web-server (N-copy). However, this work does not consider affinities and partitioning, and only examines one in-memory workload (i.e., their 100-MB file-set fits into the file-system cache). Their results show the N-copy server performs best, implying a performance gap between N-copy and multiprocessor servers [34, p. 247].

Choi *et al.* [10] use a simulator compare several web-server architectures. They conclude the server's memory footprint is important in determining its performance, especially for larger numbers of CPUs. Hence, when using an

N-copy approach on multiprocessors, servers with separate address-spaces scale poorly compared to servers with shared address-spaces. They also found contention is a problem in some servers for larger numbers of threads. However, their server architectures and simulator do not consider affinities and partitioning, scalability (outside of file-cache locking), cache consistency, or other factors in a real system.

Upadhyaya *et al.* [28] developed Aspen, a language for building parallel servers. The performance of Flash, Haboob and a server developed using Flux [9] is compared with a pipelined server developed using Aspen. The servers are implemented using different languages and some do not take advantage of sendfile to eliminate copying. Their goal is to show how Aspen can be used to easily implement parallel servers with comparable performance to existing servers.

Voras and Zagar [32] compare the multiprocessor performance of SPED, SEDA, AMPED and SYMPED architectures on their mdcached application, a memory database. Their application is a memory database rather than a web server, the largest response size is 103 bytes, and the server and clients execute on the same machine (they state this approach affects their results). Therefore, it is difficult to apply their findings to web servers.

Complementary to improving multi-core web-server performance is improving multi-core operating-system performance, which may indirectly improve concurrent application performance. A recent study by Boyd-Wickizer *et al.* [7] improves Linux kernel scalability and shows performance benefits for Apache (among other applications). A simple workload is used for performance evaluation, where all clients have non-persistent connections to repeatedly request a single 300-byte file served from the file-system cache. While performance improvements over the original Linux kernel are substantial and performance scales reasonably well with the number of cores, they only achieve a server throughput of around 2,000 Mbps using 48 cores over a 10 gigabit NIC. This low throughput is because the card's internal receive-packet queue overflows due to the large number of client packets. With a larger file, they saturated the 10 gigabit network at a "low core count". This result is consistent with our findings: a small number of cores can provide very high throughput. Another recent study by Song *et al.* [27] takes a different approach to improving operating-system scalability. They create clusters of separate Linux kernels running in different virtual machines on the same physical machine. They provide a system-call virtualization-layer above the operating systems to give applications the illusion each is running on a single operating system. Their system (Cerberus) is meant to reduce contention on shared resources by keeping the number of processors used to run each operating system small. Using Cerberus they report significantly better scalability for Apache over the standard Linux kernel. For Apache version 2.2.9 with Apache Bench as a load generator to request 45 byte files [26], their system achieves a peak total throughput of 4,000 requests per second per core using 16 cores, which translates to an aggregate throughput of only 23 Mbps. In these studies, the focus is operating system scalability versus web-server performance. In particular, their web-server experiments have low throughput with a large number of cores and a simple workload. In contrast, our web-server experiments try to achieve high throughput with a small number of cores and a more realistic workload. The extension to our work is to start with high-throughput servers and then to scale the hardware and operating system while correspondingly scaling server throughput.

Finally, Veal and Foong [29] and RouteBricks [11] describe the importance of having scalable hardware as well as software. The RouteBricks and Boyd-Wickizer *et al.* papers both point out the importance of partitioning, and each makes modifications to the Linux kernel to partition the multiple queues for the 10 gigabit NICs to different cores.

3. Experimental Environment

The experimental environment consists of eight client machines and a single server. A client machine contains two 2.8 GHz Xeon CPUs, 1 GB of RAM, a 10,000 RPM SCSI disk and four one-gigabit Ethernet cards. A client machine runs a 2.6.11.1 SMP Linux kernel and two copies of the workload generator, which permits each client load-generator to run on a separate CPU. The server machine contains two quad-core E5440 2.83 GHz Xeon CPUs, 4 GB of RAM, two 10,000-RPM 146-GB SAS hard-drives and ten one-gigabit Ethernet ports. Four of the ports are on-board, four more are from a quad-port Intel PRO/1000 PT PCI-E card and the remaining two are from a dual-port Intel PRO/1000 PT PCI-E card.

To achieve maximum performance, the server is configured with 4 cores on a single CPU, eliminating communication issues among CPUs. To examine overheads due to inter-CPU communication, experiments were conducted with 2 cores $\times$ 2 CPUs hardware configuration instead of the normal 4 cores $\times$ 1 CPU. For all servers in Section 6, the alternate configuration is only marginally slower, indicating any sharing in the servers does not result in a performance problem despite the fact the hardware utilizes bus communication rather than faster interconnects available in newer hardware.

The clients, server, network interfaces and switches have been sufficiently provisioned to ensure the network and clients are not the bottleneck. Over provisioning required multiple gigabit Ethernet-interfaces per machine organized into separate subnets, allowing for explicit load balancing of requests. Eight subnets are used to connect the server and client machines. Each client runs two copies of the load generator, with each copy using a different subnet to simulate multiple users sending requests to and getting responses from the web server. The subnets are distributed so the clients are equally spread over the eight interfaces available on the server. Hence, the clients and server communicate using fast, reliable network links. Based on a netperf experiment, the server achieved throughput of 7,500 Mbps. Our

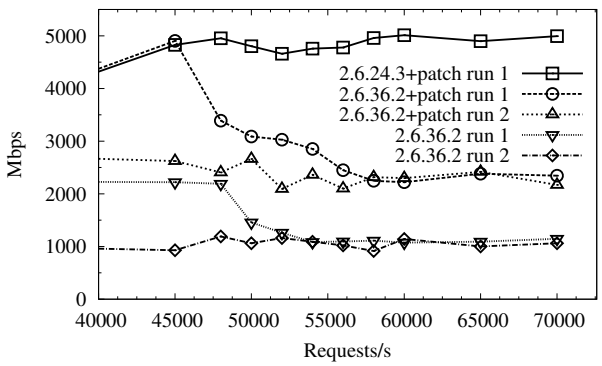

Figure 1. Throughput of different kernel versions - 2 GB

best web-server experiments are 17% below this throughput, indicating there is ample network and bus headroom.

The server runs a 2.6.24.3 SMP Linux kernel in 32-bit mode. However, a page-caching problem was discovered in this kernel resulting in poor disk performance. This problem is discussed in detail in [15, § 2.3], but the Linux kernel-patch was developed as part of the work for this paper. The patch improves disk throughput from approximately 11,000 blocks-in per second (1 block = 1024 bytes) for non-blocking sendfile and 20,000 blocks-in per second for blocking sendfile to approximately 28,000–30,000 blocks-in per second for *both* non-blocking and blocking sendfile. As well, the patch reduced the variation in throughput for repeated experiments. Therefore, the performance of all experiments requiring disk-I/O improved with the patch. Kernel 2.6.24.3 is also run in 32-bit mode as the physical address extension (PAE) and 64-bit modes resulted in lower performance, and PAE experiments did not yield repeatable results. As a result, available memory is limited to 4 GB. While 64-bit mode supports more memory, there is no expectation of better performance, and adding more memory only postpones when disk I/O occurs. Our experimental results are still relevant if the workload, bus, and memory increase proportionally.

As work progressed, newer kernels were examined, but each introduced new problems and/or instabilities with respect to this work. For example, kernel version 2.6.**36**.2 (released December 2010) has I/O performance problems with workloads having significant disk I/O (the problem exists back to at least version 2.6.32, released December 2009), but did have slightly better performance with our in-memory workload (10% faster) resulting from more efficient handling of soft-IRQs. Figure 1 shows some representative experiments, similar to the experiments in Section 8, solely to illustrate the I/O problem found in recent kernels. The graph compares throughput obtained with our patched 2.6.**24**.3 kernel, the recent 2.6.**36**.2 kernel with the same patch applied and an unpatched 2.6.**36**.2 kernel. The graph shows the patched 2.6.**36**.2 kernel has significantly higher throughput at the given rates than its unpatched counterpart; however, throughput drops substantially for higher request rates. Repeating the experiment shows an example of the large instability that occurs, demonstrating a kernel problem. Similar instability occurs for the unpatched 2.6.**36**.2 kernel but with lower throughput. The instability in disk I/O performance in recent kernels made generating verifiable results impossible for the disk-bound workloads. In comparison, our patched 2.6.**24**.3 kernel is stable with significantly higher throughput. To allow comparable results between in-memory and disk-bound workloads, a single kernel version is required. These problems and our reasons for using version 2.6.**24**.3 rather than a newer kernel are described in detail in [15]. The key reasons are, its stability after patching, the time required to optimize and tune server performance, and the uncertainty and problems encountered with the newer kernels.

4. Multiprocessor Web-Servers

Along with web-server threading, the following factors contribute significantly to high performance on a multiprocessor: CPU affinities (IRQ/process), memory footprint, shared processing, and shared memory/contention. As discussed in Section 2, using CPU affinities can improve performance, with partitioning dominating web-sever design. When there is memory pressure, a smaller web-server memory-footprint reduces the amount of disk I/O by making more memory available to the file-system cache. Shared processing can reduce the number of system calls and computational duplication, though potentially at the cost of additional contention.

Assuming a balanced load, *partitioned N-copy* achieves good performance for existing server architectures by running multiple copies of a server on a multiprocessor, partitioning the NICs, network interrupt-handling, and server. An example of N-copy partitioning in our environment (8 subnets, 4 CPUs, 4 server copies) has affinities set so $\text{CPU}_i, i = 0..3$ handles IRQ processing for subnets $2i$ and $2i+1$, and server copy S_i handles requests on subnets $2i$ and $2i+1$, and its kernel threads have CPU affinity i. Hence, each server copy executes on a single CPU and handles the two distinct subnets associated with that CPU. In general, N-copy should give excellent performance [34] by eliminating data sharing among CPUs at the application level and good CPU-cache utilization due to consistent partitioning of subnets in the OS and application. However, N-copy can be inefficient due to memory duplication and the inability to consolidate certain user and kernel operations through sharing across processes. The inefficiency varies depending on the type of server.

An alternative to N-copy is to incorporate partitioning into existing web-server architectures, resulting in shared-memory server-architectures. These servers can take advantage of techniques to improve performance involving sharing, such as reducing memory footprint (e.g., by sharing the application cache-table) and aggregating operations (e.g., by sharing file descriptors so a single call to epoll within the server process replaces per process polling in N-copy). As well, balancing the load across processes is not required. Fi-

nally, collecting global information for secondary features like cache management, client preferences, or service throttling does not require communication among processes. This work examines these benefits and how they increase server throughput by developing two web-servers: μserver shared-SYMPED (event-driven), and WatPipe (pipelined).

In [24], we extended the μserver SYMPED server to shared-SYMPED using a single application cache-table shared among the server processes to reduce data duplication. This cache-table contains only open file-descriptors and headers to efficiently support sendfile. For this paper, μserver shared-SYMPED is extended for multiprocessor execution by: 1) replacing the single listening-socket with separate sockets for each subnet; 2) partitioning processes and subnets (like N-copy), but all processes share an application cache-table across CPUs; 3) replacing the single global lock protecting the shared application cache-table with two-tiered locking to increase parallelism, where a reader-writer lock provides simultaneous file-name lookup (reads) but mutual exclusion for adding new file names (write), and a mutex lock per cache entry protects updating file data.

In [24], we developed a pipelined server, WatPipe, where each stage handles a portion of the processing of an HTTP request. Specifically, the WatPipe implementation consists of 5 stages, with each stage serviced by a separate pool of threads (implemented by Pthreads). The Accept stage accepts connections, the Read Poll stage uses an event mechanism to poll for read events, the Read stage reads and parses incoming HTTP requests, the Write Poll stage uses an event mechanism to poll for write events, and the Write stage performs the actual writes. For this paper, WatPipe is extended for multiprocessor execution by introducing queues to communicate between stages when explicit communication is required. Synchronization and mutual exclusion is required when communicating between stages and when accessing global data (e.g., open file-descriptor cache). As well, the Read and Write stages of the pipeline are partitioned. Each reader or writer thread has its affinity set to a particular CPU and only handles requests from the subnets associated with that CPU. To reduce contention, there is a separate queue per CPU for the Read and Write stages of the pipeline. The number of writer threads is one of the parameters that is varied to achieve the best performance. Having multiple threads performing writes allows server processing to continue even when a thread is blocked waiting for disk I/O, and it takes advantage of the multiple cores. Finally, a separate listening socket is created for each subnet with one acceptor thread per subnet, reducing contention in the server and OS. There are still only two threads handling polling across all subnets, one for read events and one for write events. The Acceptor, Read, and Write Poll threads have no affinities set so they are free to execute on any CPU. As WatPipe shares a single application cache-table across CPUs, its cache-table locking is modified in a fashion similar to μserver shared-SYMPED.

While the extended WatPipe seems similar to N-copy, it has some significant differences. Aside from the separate per-CPU entry-queues in the Reader and Writer stages of the pipeline, the data structures used to track connections and requests are shared and there is a single, shared application cache-table. Hence, the implementation has a smaller memory footprint than the N-copy version. Some threads are allowed to float across CPUs (acceptor, Read and Write Poll), giving the scheduler flexibility to perform some load balancing to better handle small variations in load [14, pp. 140–141]. Finally, some activities are handled by a single thread instead of having a separate thread per subnet or CPU.

To ensure a fair performance comparison among server architectures, all are implemented using the following best practices: 1) multi-accept, i.e., draining the socket's accept-queue on an event notification for a listening socket, 2) an efficient event notification mechanism (edge-triggered epoll), 3) zero-copy sendfile, 4) socket corking and no-delay to ensure small replies are sent in one packet, 5) caching open-file descriptors to avoid file-system overheads and contention, 6) caching reply headers to avoid generating them for each reply, 7) minimizing dynamic allocation (malloc/free) by pre-allocating data structures, 8) preforking processes/threads, 9) minimizing the number of kernel threads, 10) minimizing user-level processing to allow more time for kernel operations. To minimize implementation bias the servers are made as consistent as possible: WatPipe was developed from the μserver C code-base, except for small sections written in C++, sharing many common components, e.g., application cache-table of file descriptors and HTTP headers; as well, all compilations and runs use consistent options.

5. Experimental Methodology

The experiments measure throughput of the server architectures. The set of client requests and the number of CPUs is constant across experiments; server architectures and memory is varied (less memory results in disk I/O). To achieve the best server throughput, hundreds of experiments were run across a range of server parameters to *tune* each server.

The clients run httperf [20] to request a set of static files. The httperf load generator is used with session files to simulate a large number of users and to implement a *partially-open loop system* [25]. This permits httperf to produce overload conditions [3], generate multiple requests from persistent HTTP/1.1 connections, and include both active and inactive off periods to model browser processing times and user think times [4].

There are 21,600 files that could be requested across 650 directories, occupying about 2.2 GB, distributed over the server's two hard drives to achieve high throughput for disk I/O. The experiments are run with 16 clients, each running a copy of httperf (one copy per CPU), requiring a set of 16 log files with requests conforming to a Zipf distribution. A client times-out if the server does not complete a request

% Requests	10	30	50	70	80	90	**95**	100
Memory (MB)	0.5	1.5	8.4	12.2	20.1	94.3	**126.5**	2,291.6
File Size (B)	409	716	4,096	5,120	7,168	40,960	**51,200**	921,600

Table 1. Cumulative memory for requests by file size

within 10 seconds. 10 seconds is chosen because: 1) in Windows XP, the TCP/IP stack is tuned to wait 9 seconds to establish a connection before timing out, 2) based on user studies, Nielsen [22] suggests 10 seconds is the upper limit on acceptable response. Newer studies [17] suggest this value may be lower for certain types of sites.

An experiment consists of running a server with request rates ranging from 25,000 to 70,000 requests per second; each rate takes about 5 minutes to complete. There are 2 minutes of idle time between rates and between experiments to allow connections in the TIME-WAIT state to clear. Server throughput is measured both at peak and after saturation (i.e., after peak). Peak indicates the level of client requests the server can handle and after peak indicates if a server degrades gracefully (i.e. is capable of handling slashdotting). The performance metric of concern for users is response time. Although the graphs are not shown, the response times for all the servers average 1–2 seconds at peak loads.

Two workload scenarios are created by reconfiguring the server with different amounts of memory: 4 GB and 2 GB. While the workload does not change, for ease of reference these 2 scenarios are called the 4 GB and 2 GB workloads, respectively. The two workloads correspond to in-memory (4 GB) and disk-I/O (2 GB). Due to the Zipf distribution, only a small amount of memory is needed to service a significant percentage of requests. Table 1 shows the cumulative memory required to satisfy the specified percentage of requests; e.g., 95% of the requests come from 126.5 MB of the file set and 95% of the requests are for files less than or equal to 51,200 bytes. Interestingly, with 2 GB of memory, significant disk-I/O occurs (28-30 MB/sec across both disks) due to high throughput resulting from multiple processors.

For each server and workload combination, rigorous independent tuning was conducted to determine the best performance for each server. Proper tuning is critical to attain best server performance and no single tuning achieves the best performance for all servers. As the number of tuning parameters is large, only the most important subset of parameters is selected: maximum number of simultaneous connections supported by the server, level of concurrency, and blocking versus non-blocking sendfile. (Blocking sendfile and non-blocking sendfile refer to whether a socket is in blocking or non-blocking mode when sendfile is called.) After selecting the type of sendfile, a range of values for both the maximum connections and the level of concurrency is chosen. In order to see the effect of each individual parameter change, an experiment is run for the cross product of each parameter combination of the two ranges. Ranges are chosen to be sufficiently large so that the full spectrum of performance is covered. Only the best tuning-configurations for each server are analysed (see [14] for details on all tuning experiments).

Tuning also involved *verifying* the servers based on two criteria. First, preliminary experiments are run to verify server correctness (i.e., no bugs) by having each client compare the bytes returned by the server with a copy of the file requested. Due to the large overhead, correctness verification is disabled during the performance experiments. Second, quality of service is checked to ensure the server achieves an acceptable response across the range of client requests, i.e., it is not cheating by explicitly rejecting specific requests. The criteria used to establish this range ensures files of differing sizes are equally serviced; otherwise, a server can ignore certain requests to achieve performance benefits such as higher throughput or lower response times. These criteria focus on the percentage of requests that time out both cumulatively across all files and for each file size. Client requests that time out before being accepted or read by the server are not counted because the server has not seen the request. Note, since the final server comparisons are based on throughput, servers that process fewer connections (seen or unseen) will have lower throughput. While client timeouts are permitted across all file sizes, verification ensures that each size receives a reasonable level of service. The criteria are: the maximum percentage of timeouts for all files does not exceed 10%; the timeout percentage for each file size is below a certain threshold: 5% for an individual client and 2% across all clients; for each file size, the timeout percentage is not larger than the mean timeout percentage of all files plus a threshold: 5% for an individual client and 5% across all clients. The quality-of-service check does not affect throughput as it is run after the experiment completes.

6. 4 GB Workload

This section considers the performance of web-server architectures when the entire file set fits into the file-system cache by configuring the server with 4 GB of memory; however, the kernel, required daemons, and hardware mapped devices, leave only 3.6 GB available. Eliminating memory pressure highlights the multiprocessor characteristics of the architectures without focusing on disk I/O.

Figure 2 presents the best performing configuration for a selection of the server-architecture implementations (see [14] for details on all servers): μserver N-copy non-blocking SYMPED, N-copy blocking WatPipe, μserver non-blocking shared-SYMPED with partitioning, non-blocking WatPipe and blocking WatPipe. In addition, the μserver non-blocking SYMPED server without partitioning is included (network-interrupt partitioning is performed, but only a single listening socket is used) to contrast its performance with the partitioned servers. The blocking SYMPED and blocking shared-SYMPED experiments (N-copy and non-N-copy) are excluded as their memory footprint is too large, preventing the file-system cache from storing the entire file-set, resulting

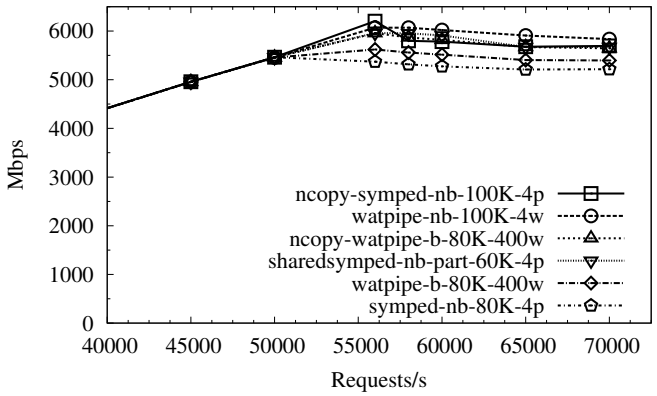

Figure 2. Throughput of different architectures - 4 GB

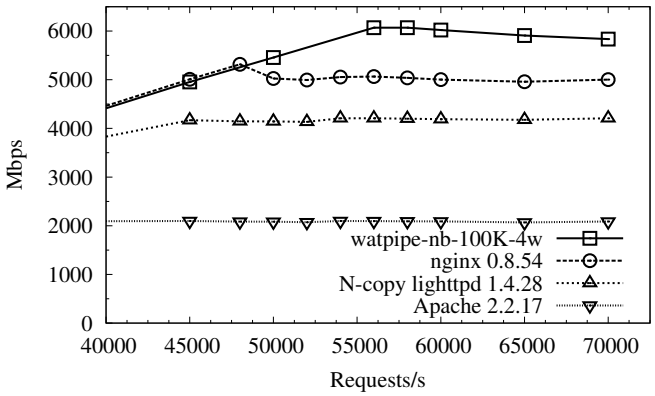

Figure 3. Comparison with open-source servers - 4 GB

in disk I/O. The legend in Figure 2 is ordered from the best performing server at the top to the worst at the bottom. Lines are labelled by server name, maximum number of connections, and processes or writer threads, e.g., ncopy-symped-nb-100K-4p is μserver N-copy non-blocking SYMPED with 100,000 connections and 4 processes. The two numbers represent the cumulative total across all copies of the server with the values distributed equally across the server copies, i.e., there are 4 server copies, one per CPU, and each copy is run with 1 process and 25,000 connections. Peak server throughput varies by 739 Mbps (5,463–6,202 Mbps). The key observation is that well implemented and tuned versions of these architectures perform well (high peak and degrade gracefully after saturation). Between the best performing versions of the server architectures, the difference is only 2% at peak, and across all servers the difference is 14%.

To better understand the performance of the servers, the best configuration of each server is profiled. Data is gathered by running OProfile and mpstat during an experiment, using a load of 56,000 requests per second; the peak performance for most of the servers. At this request rate, even with the overhead of profiling, all the servers pass verification. As no unnecessary programs or services are running on the machine during an experiment, all profiling samples can be legitimately attributed to the server's execution, including those in kernel and library code. Performance data for these experiments are summarized in Table 2. (Throughput values in the tables are incomparable due to inconsistent OProfile overheads across servers.)

In the table, each server is represented by a separate column, and its performance data is divided into three sections. The first section gives the server architecture, the configuration parameters, and the server performance in terms of both reply rate and throughput in megabits per second. The label "s-symped" means shared-SYMPED, and "part" means a single shared-SYMPED server with process affinities set so that its processes can be partitioned similar to N-copy. The second section is a summary of OProfile sampling data, consisting of the percentage of samples occurring in a particular function. The data for these functions are grouped into the Linux kernel (vmlinux), Ethernet driver (e1000), application (user space) and C library (libc). All remaining functions fall into the "other" category, which mostly represents OProfile execution. The last section contains mpstat sampling data (every 5 seconds), where the data is an average of the sampled values. The row labelled "softirq" is the percentage of time spent servicing software interrupts. Only values where there is a large difference among the servers are discussed.

As expected, with no memory pressure, N-copy servers provide an upper bound on performance. However, unlike previous work [34], the non-N-copy servers achieve performance close to their corresponding N-copy server, especially for the non-blocking servers with partitioning. Since the best performance occurs with 1 process per CPU, N-copy SYMPED and N-copy shared-SYMPED are equivalent. When comparing N-copy and non-N-copy servers, contention related to sharing data across CPUs is a factor. Yet, for the non-blocking servers, which only require a small number of threads, given appropriate data structures and locking (e.g., using readers/writer locks in the application cache-table), moving from N-copy to non-N-copy has little effect even when data is shared across CPUs. On the other hand, moving from N-copy to non-N-copy has a larger effect for the blocking servers as many kernel threads are sharing data structures across CPUs. Specifically for WatPipe, these sharing overheads result in a drop of 6% in peak performance from N-copy to non-N-copy blocking WatPipe versus 2% from N-copy to non-N-copy non-blocking WatPipe. (The N-copy non-blocking WatPipe server is not in Figure 2 but its peak is 6196 Mbps.) The larger performance difference for blocking WatPipe is a result of the large number of kernel threads sharing data across CPUs because all other server differences are consistent between the blocking and non-blocking versions. WatPipe is able to achieve excellent throughput despite additional overheads from sharing data across CPUs because it does not partition all stages (e.g., polling) or pin all threads to specific CPUs. Therefore, by allowing some threads to float, the scheduler has some flex-

EXPERIMENT	Server	**userver**	**WatPipe**	**userver**	**userver**	**WatPipe**	**WatPipe**
	Arch	**symped**	**pipeline**	**symped**	**s-symped**	**pipeline**	**pipeline**
	Write Sockets	**non-block**	**block**	**non-block**	**non-block**	**non-block**	**block**
	Max Conns	**100K**	**80K**	**80K**	**60K**	**100K**	**80K**
	Processes/Writers	**4p**	**400w**	**4p**	**4p**	**4w**	**400w**
	Other Config	**N-copy**	**N-copy**		**part**		
	Reply rate	47,474	48,116	43,329	49,717	49,886	45,666
	Tput (Mbps)	5,666	5,729	5,176	5,944	5,944	5,448
OPROFILE	vmlinux total %	81.70	81.65	81.38	81.94	79.33	80.84
	e1000 total %	11.23	10.45	11.90	10.82	10.91	10.36
	user-space total %	5.33	3.95	5.01	5.49	5.44	4.71
	libc total %	0.90	1.10	0.88	0.89	1.16	1.06
	other total %	0.84	2.85	0.83	0.86	3.16	3.03
MPSTAT	softirq %	58	58	61	59	58	56

Table 2. Server performance statistics gathered under a load of 56,000 requests per second - 4 GB

EXPERIMENT	Server	**WatPipe**	**WatPipe**	**userver**	**userver**	**WatPipe**	**WatPipe**
	Arch	**pipeline**	**pipeline**	**s-symped**	**s-symped**	**pipeline**	**pipeline**
	Write Sockets	**non-block**	**block**	**non-block**	**block**	**non-block**	**block**
	Max Conns	**50K**	**60K**	**50K**	**50K**	**50K**	**60K**
	Processes/Writers	**100w**	**500w**	**60p**	**300p**	**100w**	**500w**
	Other Config	**N-copy**	**N-copy**	**part**	**part**		
	Reply rate	37,902	41,161	38,253	38,139	37,489	39,951
	Tput (Mbps)	4,511	4,905	4,560	4,562	4,461	4,756
OPROFILE	vmlinux total %	82.50	81.37	83.35	81.81	81.40	80.24
	e1000 total %	8.81	9.49	9.13	9.38	8.83	9.38
	user-space total %	3.90	4.18	4.41	4.94	4.69	5.08
	libc total %	0.90	1.02	0.78	1.15	0.93	1.03
	other total %	3.89	3.94	2.33	2.72	4.15	4.27
VMSTAT	waiting %	23	7	21	17	20	5
	file-system cache (MB)	1,519	1,476	1,530	1,364	1,556	1,522
	blocks-in/sec	29,620	30,399	28,523	31,748	27,908	27,553
MPSTAT	softirq %	43	49	47	45	42	49

Table 3. Server performance statistics gathered under a load of 56,000 requests per second - 2 GB

ibility to perform a small amount of load balancing to better handle small variations in load.

The biggest factor affecting performance is whether the server supports affinities and partitioning, which reduces overheads, especially related to networking. When the kernel spends more time processing network packets, as indicated by the e1000 and softirq values, it is due to either higher throughput (more processing time is required to handle more packets per second) or the inefficient handling of packets. When comparing the poor performing non-partitioned μserver non-blocking SYMPED server (throughput of 5,176 Mbps) and the much better performing μserver N-copy non-blocking SYMPED server (throughput of 5,666 Mbps), the only differences between them are affinities and partitioning. Despite these significant differences in throughput, they spend a similar amount of time executing e1000 (11.9% versus 11.2%) and softirq code (61% versus 58%) to process network packets. Hence, μserver N-copy non-blocking SYMPED is more efficient and processes a larger number of network packets than non-partitioned μserver non-blocking SYMPED. This shows the increased overheads incurred as a result of not partitioning the processes, subnets and CPUs.

Overall, for in-memory workloads, non-blocking servers achieve the best performance as they require few kernel threads, and incur less overhead.

7. Comparison with Other Servers

According to Netcraft's October 2011 survey [21], the market share for the top servers were Apache 58.9%, Microsoft 12.5%, nginx 11.3%, Google 8.1%, while lighttpd (used here) was popular and appeared in the December 2010 survey. nginx has increased in popularity due to its good performance on static workloads when compared with Apache, illustrating the importance of this workload. To demonstrate that our servers provide good performance relative to other available servers, three of these web servers are run in our environment (nginx 0.8.54, lighttpd 1.4.28 and Apache 2.2.17). Each server is configured for performance, to use a minimum footprint, and tuned to find the best configuration for the environment and workload being used. (The Apache MPM worker module was selected based on reported best performance, and tuned for our environment.) Support in lighttpd for the server.max-worker configuration parameter and partitioning are weak resulting in poor shared-memory performance so it is run using an N-copy configuration. Furthermore, lighttpd had to be modified to increase open files and connections as the original limit was 65,536 open files and roughly half as many simultaneous connections.

Figure 3 compares our best performing non-N-copy server from the 4 GB experiment with the three open-source

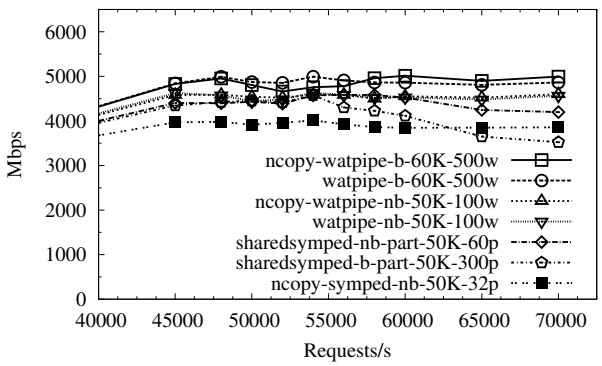

Figure 4. Throughput of different architectures - 2 GB

servers. This experiment is used solely to place our servers in context with other well-known servers. It would be extremely difficult to draw any conclusions about server architecture given the differences in code base among the servers. Ranking the servers from best to worse: watpipe-nb-100K-4w, nginx, lighttpd, and Apache, with peaks of 6070, 5316, 4207, 2097 Mbps, respectively, an overall difference of 65%. All servers perform well after peak. Apache's poor performance under high load (also shown by [23, 31, 33]) results from the thread-per-connection architecture, using a 1-to-1 threading model (Pthreads), which incurs high overhead due the large number of kernel threads. (We believe the thread-per-connection architecture is not viable for high-performance servers without a highly-efficient M-to-N user-level thread package for multiprocessors [31].) As well, the Apache footprint leaves only 1.67 GB of memory for the file-system cache, resulting in disk I/O, while the other servers cache all files (2.2 GB).

8. 2 GB Workload

This section considers the performance of web-server architectures when the entire file set does not fit into the file-system cache (i.e., there is memory pressure) by configuring the server with 2 GB of memory. While the memory pressure seems low relative to the 2.2 GB file set, given the high request rates, even a small percentage of requests requiring disk I/O results in significant disk activity; hence, these experiments have lower throughput.

Figure 4 presents the best performing configuration for each server-architecture implementation (more details on all servers can be found in [14]). The legend in Figure 4 is ordered from the best performing server at the top to the worst at the bottom. Peak server throughput varies by about 20% (4,012–5,012 Mbps), a range of 1,000 Mbps. Performance of these servers can be loosely grouped into pairs, except for the bottom server. The top two performers are N-copy blocking WatPipe and blocking WatPipe, which have approximately the same peak throughput of 5,000 Mbps and the same overall performance. The remaining server pairs have approximately the same peak throughput of 4,600 Mbps, but different performance after saturation. The next two servers are the non-blocking versions of those servers, which have the same performance with only a small decline after peak, approximately 9% lower than N-copy blocking WatPipe. The next two servers (shared-SYMPED non-blocking and blocking with partitioning) have the largest decline after peak, approximately 19% and 42% lower than N-copy blocking WatPipe, respectively. The last server (μserver N-copy non-blocking SYMPED) has the lowest peak throughput of 4,012 Mbps, approximately 20% lower than N-copy blocking WatPipe with stable performance after peak.

Table 3 shows the same OProfile data as the 4 GB workload (excluding N-copy μserver non-blocking SYMPED), plus a third section for vmstat sampling data (every 5 seconds), consisting of data about processes, memory, I/O, CPU activity, etc., where the data is an average of the sampled values. There are three rows in the vmstat data labelled: "waiting %", the CPU time spent waiting for IO, "file-system cache", the average size of the Linux file-system cache in megabytes, and "blocks in/sec", the average number of blocks read per second. Note, a non-zero I/O wait value indicates that the profiling data must be scaled because it only accounts for time when the CPU is executing, so it does not include I/O wait. For example, if the I/O wait is 30%, then the profiling data still adds up to 100% but only covers the 70% of the time the CPU is in use.

All servers have a non-zero I/O wait value for their best configuration, indicating an opportunity to improve performance by utilizing unused CPU time. One way to eliminate I/O wait is to increase connections and/or kernel threads, allowing the server to overlap processing with disk-I/O. Unfortunately, increasing these parameters also increases execution overheads (e.g., lock contention) and the memory footprint of the server. These changes cause the I/O wait to decrease or increase, but for all the servers it causes throughput to decrease as both the overheads and the memory footprint of the server increase (see [14] for more details).

While there is a performance difference resulting from blocking/non-blocking sendfile (discussed next), differences resulting from memory usage among the servers using the same kind of sendfile (blocking versus non-blocking) are discussed first. An important aspect of memory usage is the Linux file-system cache used to cache data from disk, including meta-data, directory information and file data. The size of the file-system cache is determined by the amount of memory *unused* by the kernel and server, which dictates the amount of disk I/O required during an experiment (i.e., a larger file-system cache can hold more file data so fewer requests require file data to be read from disk). The servers in Table 3 achieve their best performance with similarly sized file-system caches, implying similar memory footprints. However, the configuration parameters at which the servers achieve their best performance is not consistent among the servers. Specifically, the servers with lower

throughput are configured with fewer kernel threads due to less efficient memory scaling.

In detail, the memory allocated by each server when adding kernel threads is: $M_{sympd} = c \times 810 + p \times (7{,}667{,}536 + s)$, $M_{sharedsympd} = c \times 806 + p \times (216,100 + 8 \times c + 20 \times p + s) + N \times 8,404,368$, and $M_{watpipe} = c \times 934 + N \times 9,798,888 + p \times s$, where c is the number of connections, p is the number of kernel threads, N is the number of copies ($N = 1$ for non-N-copy servers), and s is the thread stack-size (between 16 KB and 64 KB is typically accessed, with 32K used in this analysis). Note, these estimates are an upper bound as real memory usage is often lower because it depends on pages accessed versus pages allocated, but this is difficult to model. The increase in memory footprint for all the servers is similar as connections are added (810, 806, 934 bytes), though WatPipe requires about 15% more memory per connection. However, the footprint of the servers scales differently as kernel threads are added. For the SYMPED servers, every kernel thread is a completely separate process, so its memory footprint grows quickly, resulting in lower throughput; SYMPED grows by about 7.3 MB per additional process, with around 6.4 MB of that coming from the separate application cache (open file-descriptors and file headers) in each process. For shared-SYMPED, the processes share a single application cache, so its memory footprint grows less quickly; shared-SYMPED grows by about .6 MB per additional process. The size of WatPipe increases only by about .03 MB for each writer as the address space is shared. These values are small per kernel thread, but with 500 or more kernel threads, they can result in large differences in memory footprint, e.g., with 500 kernel threads the increase is 3,672 MB for SYMPED and 314 MB for shared-SYMPED and 16 MB for WatPipe. Therefore, WatPipe can add writers with only a small increase in memory footprint from the thread stacks. With the additional kernel threads, WatPipe can overlap more computation with blocking disk I/O, reducing I/O wait, resulting in higher throughput. Comparing the N-copy and non-N-copy servers (excluding SYMPED, which is already N-copy), the N-copy servers have only a small increase in memory because it is proportional to the number of cores, i.e., for the 4-copy shared-SYMPED experiment the increase is 24 MB per copy, and for the 4-copy WatPipe experiment it is 28 MB per copy. For thread-per-connection servers, p is equal to c, so memory for stacks is $c \times s$, e.g., for 50,000 connections 1,563 MB is needed just for stacks, which should adversely affect performance. For large SMP computers, increasing total memory decreases the effect of these differences, but increasing CPUs requires more kernel threads to support higher loads, using more memory.

An important clue to understand why the blocking sendfile versions of WatPipe perform so well is found in the amount of time each spends waiting on I/O. In Table 3, the vmstat line "waiting %" shows these servers spend significantly less time waiting for disk I/O than the corresponding non-blocking servers. However, since the blocking servers require more threads and as a result have larger memory footprints, the expectation is that they could spend more time waiting for I/O. Part of the lower I/O wait times can be attributed to additional overhead incurred by the blocking servers. Low I/O wait combined with higher throughput indicate that blocking sendfile accesses the disk more efficiently than non-blocking sendfile for this workload. After analysing the disk request patterns for N-copy non-blocking and blocking WatPipe, some interesting differences were found. The blocking server makes disk-I/O requests on fewer distinct files, and for larger files, these requests tend to be contiguous, allowing the server to take advantage of file read-ahead caching. The result is the blocking server makes fewer disk requests overall compared to the non-blocking server and is hence able to service more requests. Therefore, the remaining part of the lower I/O wait can be attributed to more efficient disk access for the blocking servers.

The difference between the two servers is the duration over which file data in the file-system cache is accessed. As this cache is managed using an LRU algorithm, when there is memory pressure in the system, blocks accessed together in time are likely to be evicted together. With blocking sendfile, each kernel thread blocks while sending an entire file, so the file blocks are usually accessed over a relatively small period of time. Subsequently, these file blocks age together, and hence, are likely to be evicted together. With non-blocking sendfile, each kernel thread interleaves the sending of large files with the sending of many other files. (Large files are sent in chunks, where the size of each send is constrained by the space available in the socket buffer.) As a results, portions of many files have similar access times and different portions of a large file have access times different from each other. Subsequently, pieces of different files age together, and hence, are likely to be evicted together, leaving other pieces of the large file still in the cache. The benefit is realized when a file is accessed again. For blocking sendfile, if any portion of the file must be reread from disk, it is likely the entire file must be read again, which is accomplished via efficient contiguous disk-reads and better file read-ahead caching. Overall, the blocking server has better disk efficiency than the non-blocking server. Therefore, a server using blocking sendfile can have the same or higher throughput than a corresponding non-blocking server despite having a larger memory footprint.

9. Comparison Across Workloads

The previous sections highlighted a number of factors affecting server performance for a particular workload; this section provides an overview of the performance of the multiprocessor servers *across* the two workloads tested. The bars in Figure 5 represent the area under the throughput curve of the servers across the request-rates tested for the two workloads. The areas are normalized based on the largest server

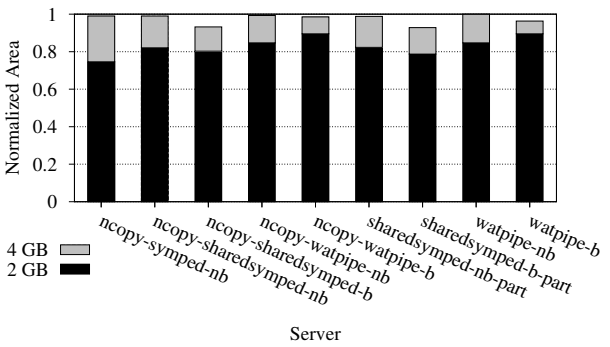

Figure 5. Server performance across workloads

area, non-blocking WatPipe. While the area gives a simplified view of overall performance, it deemphasizes differences in peak throughput. The key observation is that well implemented and tuned versions of the architectures perform well across workloads with only small differences.

In Figure 5, the 4 GB bars are all higher than the 2 GB bars, indicating that as memory pressure in the system increases, the throughput of the servers decreases. When there is no memory pressure, a pattern emerges: the non-blocking servers require few kernel threads, resulting in low overheads and slightly higher throughput, and the blocking servers require more kernel threads, resulting in higher overheads and slightly lower throughput. When there is memory pressure, a different pattern emerges: memory footprint and disk efficiency are two important factors determining server performance. In this case, the blocking Watpipe servers offer better performance as they can scale to a large number of kernel threads with only a small increase in memory footprint by using a shared-address space. Due to the dissimilar requirements for each pattern, no single server's performance is the best for both workloads.

10. Tuning Insights

A key to good web-server performance is tuning the server for the hardware environment and the workload being serviced. Since tuning is a time consuming task, we present some insights gained during this work: 1) For workloads serviced entirely from the file-system cache, high throughput is obtained by using non-blocking sockets and one kernel thread per core for writing (though it may perform other work, depending on the architecture). 2) The number of simultaneous connections being served is critical, especially after saturation. Too few results in a lower peak-throughput, while too many typically results in longer response times with requests for larger files taking too long. 3) For workloads with disk I/O, the key is efficiently switching to a connection that can be serviced while disk I/O occurs. For the servers we examined, this is achieved by having a sufficient number of kernel threads. Too few kernel threads can leave the CPU idle; too many reduces memory available for the file-system cache.

11. Conclusion

This work shows that, if properly implemented and tuned, N-copy, event-driven, and pipelined architectures can all perform very well on 4 cores while servicing the two static workloads examined in this paper. However, within architectures not all versions are amenable to both workloads. While no single server or configuration performed the best for all workloads, with proper tuning the difference in peak throughput among the best version of each server architecture is within 10%. The key factors affecting the performance of these architectures are memory footprint, usage of blocking or non-blocking calls to sendfile, controlling contention for shared resources (e.g., locks), ensuring utilization of processor affinities and partitioning, and supporting a large number of simultaneous connections. We have demonstrated that both non-shared-memory (N-copy) and shared-memory servers can offer competitive performance. The advantage of sharing data is a smaller memory footprint, but the trade off is increased overheads due to contention. Any server architecture with shared data must control these overheads, otherwise they become a bottleneck. However, as Internet services continue to evolve and become more complex, web servers need to support complex secondary features (e.g., service throttling), which is more easily accomplished with direct access to shared state.

The current hardware trend is SMPs with increased core counts and 10 GB Ethernet NICs. Based on recent papers [7, 11], partitioning and affinities continue to be necessary for high-performance (e.g., partitioning multiple queues on 10 GB network cards). Scaling to a large number of cores requires a large number of threads, which introduces a tradeoff between data-duplication and contention. Based on our experiments, a small increase in data duplication can mitigate contention and reduce inter-CPU communication, e.g., for our N-copy WatPipe experiments, data-duplication is relative to the number of cores so contention is isolated to the threads on a core. This approach can be extended by expanding data-sharing to all the cores on a CPU to reduce the amount of duplication, which could lead to an increase in throughput when there is memory pressure. However, with the high throughputs obtained using small core counts in this paper, we do not believe large SMPs will be used for a single web server. It is more likely many web servers will be run on a single SMP with high core counts possibly within virtual machines. In such an environment, we believe the ability to restrict web servers to execute on a subset of cores and NICs, and to provide effective partitioning is even more important.

12. Acknowledgments

Funding for this project was provided by NSERC Canada.

References

[1] V. Anand and B. Hartner. TCPIP network stack performance in Linux kernel 2.4 and 2.5. In *Proc. of the 4th Ottawa Linux*

Symp., June 2002.

[2] Apache Software Foundation. The Apache web server. http://httpd.apache.org.

[3] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of the USENIX Symp. on Internet Technologies and Systems*. USENIX Association, 1997.

[4] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of ACM SIGMETRICS 1998*, Madison, Wis., 1998.

[5] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Operating System Design and Implementation*, 2010.

[6] A. C. Begen, T. Akgul, and M. Baugher. Watching video over the web: Part 1: Streaming protocols. *IEEE Internet Computing*, 15(2):54–63, 2011. ISSN 1089-7801.

[7] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proc. of the 9th USENIX Symp. on Operating Sys. Design and Impl.*, pages 1–16, Oct. 2010.

[8] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. In *EuroSys'06*, pages 265–278, April 2006.

[9] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A language for programming high-performance servers. In *Proc. of the USENIX Annual Tech. Conf.*, pages 129–142, 2006.

[10] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das. A multithreaded pipelined web server architecture for SMP/SoC machines. In *Proc. of the 14th international conf. on World Wide Web*, pages 730–739, 2005.

[11] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 15–28, 2010.

[12] Facebook. Open compute project. http://opencompute.org/-specs.

[13] A. Foong, J. Fung, and D. Newell. An in-depth analysis of the impact of processor affinity on network performance. In *Proc 12th IEEE International Conference on Networks*, volume 1, pages 244–250, Nov. 2004.

[14] A. S. Harji. *Performance Comparison of Uniprocessor and Multiprocessor Web Server Architectures*. PhD thesis, University of Waterloo, 2010. http://uwspace.uwaterloo.ca/-bitstream/10012/5040/1/Harji_thesis.pdf.

[15] A. S. Harji, P. A. Buhr, and T. Brecht. Our troubles with linux and why you should care. In *2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)*, July 2011.

[16] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High-performance memory-based Web servers: Kernel and user-space performance. In *Proc. of the USENIX Annual Tech. Conf.*, pages 175–188, 2001.

[17] Jupiter Research. Retail website performance: Consumer reaction to a poor online shopping experience. http://www.akamai.com/4seconds, 2006.

[18] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. of the USENIX Annual Tech. Conf.* USENIX, 2004.

[19] Microsoft. Microsoft reveals its specialty servers, racks. http://www.datacenterknowledge.com/archives/2011/04/25/-microsoft-reveals-its-specialty-servers-racks.

[20] D. Mosberger and T. Jin. httperf tool for measuring web server performance. *ACM SIGMETRICS*, 26(3):31–37, 1998.

[21] Netcraft. Oct. 2011 Web Server Survey, 2011. http://news.-netcraft.com/archives/2011/10/06/october-2011-web-server--survey.html.

[22] J. Nielsen. *Designing Web Usability*. New Riders, 2000.

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. of the USENIX Annual Tech. Conf.*, 1999.

[24] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the performance of web server architectures. In *Proc. of the 2nd ACM EuroSys Conf. on Computer Systems*, pages 231–243, March 2007. ISBN 978-1-59593-636-3.

[25] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Network System Design and Implementation*, 2006. http://www.usenix.org/events/-nsdi06/tech/schroeder.html.

[26] X. Song. Personal communication, 2011.

[27] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *EuroSys Conf. on Computer Systems*, 2011.

[28] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and exploiting concurrency in networked applications with Aspen. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 13–23, 2007.

[29] B. Veal and A. Foong. Performance scalability of a multicore web server. In *Proc. of the 3rd ACM/IEEE Symp. on Architecture for Networking and Communications Systems*, pages 57–66. ACM, 2007.

[30] VMware. Consolidating web applications using VMware infrastructure. http://www.vmware.com/files/pdf/consolidating-_webapps_vi3_wp.pdf.

[31] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 268–281, 2003.

[32] I. Voras and M. Žagar. Characteristics of multithreading models for high-performance IO driven network applications. In *AFRICON, 2009. AFRICON '09.*, pages 1–6, Sept. 2009.

[33] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 230–243. ACM Press, 2001.

[34] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proc. of the USENIX Annual Tech. Conf.*, pages 239–252, June 2003.