Comparing the Performance of Web Server Architectures

David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
db2pariag,brecht,asharji,pabuhr,ashukla @cs.uwaterloo.ca

ABSTRACT

In this paper, we extensively tune and then compare the performance of web servers based on three different server architectures. The µserver utilizes an event-driven architecture, Knot uses the highly-efficient Capriccio thread library to implement a thread-per-connection model, and WatPipe uses a hybrid of events and threads to implement a pipeline-based server that is similar in spirit to a staged event-driven architecture (SEDA) server like Haboob.

We describe modifications made to the Capriccio thread library to use Linux's zero-copy sendfile interface. We then introduce the SYmmetric Multi-Processor Event Driven (SYMPED) architecture in which relatively minor modifications are made to a single process event-driven (SPED) server (the µserver) to allow it to continue processing requests in the presence of blocking due to disk accesses. Finally, we describe our C++ implementation of WatPipe, which although utilizing a pipeline-based architecture, excludes the dynamic controls over event queues and thread pools used in SEDA. When comparing the performance of these three server architectures on the workload used in our study, we arrive at different conclusions than previous studies. In spite of recent improvements to threading libraries and our further improvements to Capriccio and Knot, both the event-based µserver and pipeline-based WatPipe server provide better throughput (by about 18%). We also observe that when using blocking sockets to send data to clients, the performance obtained with some architectures is quite good and in one case is noticeably better than when using non-blocking sockets.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.4.1 [**Process Management**]: Concurrency, Threads; D.4.8 [**Performance**]: Measurements

General Terms

Design, Experimentation, Measurement, Performance

Keywords

web servers, threads, events, scalability, performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003 ...$5.00.

1. INTRODUCTION

Several different server architectures have been proposed for managing high levels of server concurrency. These can be broadly categorized as event-driven, thread-per-connection (sometimes referred to as thread-based) or a hybrid utilizing a combination of events and threads.

Previous research [18] has analyzed the strengths and weaknesses of many of these architectures. However, more recent work [24, 8, 22, 23] has re-ignited the debate regarding which architecture provides the best performance. The most recent results conclude that the thread-per-connection Knot server [22], which is based on the scalable user-level threading-package Capriccio [23], matches or exceeds the performance of the hybrid, pipeline-based Haboob server. In this paper, we conduct benchmark experiments to compare the performance of well tuned, state-of-the-art event-based and thread-per-connection servers (the µserver and Knot), along with a hybrid, pipeline-based server (called WatPipe). WatPipe's architecture is designed to be similar to the staged event-driven architecture (SEDA) used to implement Haboob [24]. While SEDA and Haboob are written in Java and use dynamic controllers to adjust the number of threads and load shedding within each stage, WatPipe is written in C++ and does not implement any controllers.

The contributions of this paper are:

- We present two new approaches for supporting the zero-copy `sendfile` system call within the Capriccio threading library, and evaluate their performance using the Knot web server.
- The SYmmetric Multi-Process Event Driven (SYMPED) and shared-SYMPED server architectures are introduced.
- A hybrid pipeline server-architecture, based on a simplified staged event-driven (SEDA) architecture, is introduced.
- The importance of properly tuning each of the server architectures is demonstrated. We find that with the workload used in this paper, performance can vary significantly with the number of simultaneous connections and kernel threads.
- The event-based µserver and the pipeline-based WatPipe servers are found to perform as well as or better than the thread-per-connection Knot server on the workload used in this paper.
- Contrary to prior work, the results obtained using our workload show that the overheads incurred by the thread-per-connection architecture make it difficult for this architecture to match the performance of well implemented servers requiring substantially fewer threads to simultaneously service a *large* number of connections.
- In some of the architectures examined, the use of blocking sockets for writing replies is found to perform quite well.

The remainder of this paper is organized as follows. Section 2 describes background and related work. Section 3 describes our experimental environment and workload. Sections 4, 5, and 6 describe the implementation and performance tuning experiments conducted for the Knot, the µserver and WatPipe servers, respectively. In Section 7, several server configurations are compared and analyzed using the best performance of each tuned server. Section 9 explains why our findings differ substantially from previous work, and Section 10 presents our conclusions.

2. BACKGROUND AND RELATED WORK

Modern Internet servers must efficiently operate on thousands of files and connections [3]. However, operations on files and sockets may cause a server to block unless special action is taken. Therefore, strategies for dealing with blocking I/O are a key requirement in most server architectures. The main strategies are: do nothing, use non-blocking or asynchronous I/O, or delegate potentially blocking I/O operations to threads, either among separate server processes or within a single server process using multiple kernel threads.

While most Unix-like systems support non-blocking socket I/O, support for non-blocking file I/O is poor or non-existent. Asynchronous I/O (AIO) mechanisms exist for some file operations, but performance critical system calls like `sendfile` have no AIO equivalents [14]. In light of these limitations, many server architectures have adopted threading to mitigate the effects of blocking I/O. Threading is introduced via kernel processes and/or user-level thread libraries.

Threads do not prevent blocking due to I/O; they only provide possible alternative executions. For example, when one thread blocks due to I/O, other non-blocked threads may still be able to execute. This approach relies on transparent context switching of kernel threads by the operating system or user-level threads by a thread library to continue execution. Typically, a user-level thread-library attempts to avoid blocking due to I/O by wrapping appropriate system calls with library specific code. These wrapped calls check if socket I/O can be performed without blocking and only schedules user-level threads that can make those calls without blocking. This is done using an event mechanism like `select` or `poll` or, for some system calls, by putting the socket into non blocking mode and making the call directly. Additionally, many system calls that can potentially block due to file system I/O may be handed off to workers that are implemented using kernel-threads so that the operating system can context switch from workers that block due to disk I/O to workers that are not blocked. With this approach, care must be taken to ensure that enough kernel-level threads are available to handle requests made by user-level threads and access to data that is shared among threads is properly synchronized without incurring excessive overheads. An additional concern in this and other web server architectures is using enough kernel-level threads to make certain that all kernel-threads do not become blocked simultaneously.

One of the main differences among web server architectures is the association of connections with threads. In thread-per-connection servers at least one user-level thread and possibly a kernel thread is required for each connection. Pai et al. [18], refer to shared-memory thread-per-connection servers (multiple threads within a single address space) as multi-threaded (MT) and separate processes-per-connection servers (all threads of execution are in different address spaces) as multi-process (MP). In architectures that require a large number of threads, overhead due to thread scheduling, context-switching, and contention for shared locks can combine to degrade performance. In fact, architects of early systems found it necessary to restrict the number of concurrently running threads [3, 12]. Restricting the number of threads means either restricting the number of simultaneous connections (in a thread-per-connection server) or multiplexing multiple connections within each thread.

The Single Process Event-Driven (SPED) [18] architecture multiplexes all connections within a single process by putting all sockets into non-blocking mode, and only issuing system calls on those sockets that are ready for processing. An event notification mechanism such as `select`, `poll`, `epoll`, or `kqueue` is used to identify those sockets that are currently readable or writable. For each such socket, the server invokes an appropriate event-handler to process the request without causing the server to block. The SPED architecture leverages operating system support for non-blocking socket I/O. However, the SPED architecture makes no provisions for dealing with blocking file I/O operations. Therefore, alternative server architectures are required for workloads that induce blocking due to disk activity.

In this paper, we introduce the SYmmetric Multi-Process Event-Driven (SYMPED) architecture (see Section 5.1), which extends the SPED model by employing multiple processes each of which acts as a SPED server to mitigate blocking file I/O operations and to utilize multiple processors. The SYMPED model relies on the operating system to context switch from a process that blocks to a process that is ready to run.

The Flash server implements the Asymmetric Multi-Process Event-Driven (AMPED) [18] architecture, combining the event-driven approach of SPED with helper processes dedicated to performing potentially blocking file system operations. In this architecture, a single event-driven process delegates all file system I/O to helper processes, which invoke the potentially blocking operation. Processes may or may not share address spaces. Like the SYMPED architecture, the AMPED model relies on the operating system for all process scheduling. One potential drawback of the AMPED approach is the coordination and communication required between the SPED server and its helper processes.

In order to directly compare the performance of different server architectures Pai et al. [18] implemented several different architectures within the Flash server. They demonstrate that a SPED server outperforms threaded servers on in-memory workloads, and the AMPED server matches or exceeds the performance of a thread-per-connection server on disk-bound workloads.

The Staged Event-Driven Architecture (SEDA) [24] consists of a network of event-driven stages connected by explicit queues. Each SEDA stage uses a thread pool to process events entering that stage. SEDA allows stages to have private thread pools or to share thread pools among stages. The size of each stage's thread pool is governed by an application-specific resource controller, and threads are used to handle blocking I/O operations and to utilize CPUs in a multiprocessor environment. While SEDA does not use a separate thread for each connection, concurrency still requires the use of (possibly large) thread pools.

Recent work by von Behren et al. [22] argues that many of the observed weaknesses of threads are due to poorly implemented user-level threading libraries, and are not inherent to the threading model. As evidence, the authors present experiments comparing the Knot and Haboob web servers. Knot is a thread-per-connection server written in C, using the lightweight, cooperatively scheduled user-level threads provided by the Capriccio threading library. Haboob is a SEDA-based server written in Java. Each of Haboob's SEDA stages contains a thread pool and application logic responsible for handling a portion of the processing required to handle an HTTP request.

Based on an experimental comparison of Knot and Haboob, von Behren et al. conclude that threaded servers can match or exceed the performance of event-driven servers. However, they also observe that Haboob context switches more than 30,000 times per second under their workloads [22]. They point out that this behaviour is partly due to the context switching required when events pass from one SEDA stage to another. Other studies have compared the performance of different servers with Haboob and shown the performance of Haboob is significantly lower than current state-of-the-art servers [6, 19].

In this paper we experimentally evaluate, analyze and compare the performance of three different server architectures: an event-driven SYMPED (using the µserver), thread-per-connection (using Knot) and a SEDA inspired pipeline-based architecture (using WatPipe).

3. METHODOLOGY

We now describe the hardware and software environments used to conduct the experiments described in the remainder of the paper.

3.1 Environment

Our experimental environment consists of four client machines and a single server. The client machines each contain two 2.8 GHz Xeon CPUs, 1 GB of RAM, a 10,000 rpm SCSI disk, and four one-gigabit Ethernet cards. They run a 2.6.11-1 SMP Linux kernel which permits each client load-generator to run on a separate CPU. The server is identical to the client machines except that it contains 2 GB of memory and a single 3.06 GHz Xeon CPU. For all experiments the server runs a 2.6.16-18 Linux kernel in uni-processor mode. A server with a single CPU was chosen because Capriccio does not implement support [23] for multiprocessors.

The server and client machines are connected via multiple 24-port gigabit switches. On each client machine two CPUs are used to run two copies of the workload generator. Each copy of the workload generator uses a different network interface and simulates multiple users who are sending requests to and getting responses from the web server. The clients, server, network interfaces and switches have been sufficiently provisioned to ensure that the network and clients are not the bottleneck.

3.2 Workload

Our HTTP workload is based on the static portion of the widely used SPECweb99 [21] benchmarking suite. However, we address two notable problems with the SPECweb99 load-generator. The first problem is that the *closed-loop* SPECweb99 load-generator does not generate overload conditions because it only sends a new request after the server has replied to its previous request. Banga et al. [2] show this simple load-generation scheme causes the client's request rate to be throttled by the speed of the server, which is especially a problem when studying servers that are meant to be subjected to overload. The second problem is that the SPECweb99 load-generator does not adequately model user think-times, web-browser processing-times, or network delays. Instead new requests are initiated immediately after the response to the previous request.

The first problem is addressed by using *httperf* [16] and its support for session log files to create a *partially-open loop system* [20]. This permits us to produce representative workloads because requests are generated by new arrivals and by multiple requests from persistent HTTP/1.1 connections. Additionally, overload conditions are generated by implementing connection timeouts. In our experiments, the client-side timeout is 15 seconds for each request.

The second problem is addressed by including periods of inactivity in our HTTP traces. Barford and Crovella [4] have developed probability distributions to model both "inactive" and "active" off-periods. Inactive off-periods model the time between requests initiated by the user, which includes time spent reading a web page, as well as any other delays occurring between user requests. Active off-periods model the time between requests initiated by the web browser, which includes time spent parsing the initial page, subsequently sending requests to fetch objects embedded within that page, and waiting for the responses to such requests. Barford and Crovella observe that inactive off-periods (user think-times) follow a Pareto probability distribution, while active off-periods (browser think-times and delays) follow a Weibull distribution.

Recent work [13] measured browser think-times using modern Internet browsers on both Windows and Linux. Although the methodologies used are quite different, the resulting models for think-times are quite similar. Based on the observations made in these previous studies we chose an inactive off-period of 3.0 seconds and an active off-period of 0.343 seconds. Note that under our workload both the active and inactive off-periods take into account a notion of delays that can occur when communicating between a client and a server, including effects of wide area network (WAN) traffic.

Our SPECweb99 file-set contains 24,480 files, totalling 3.2 GB of disk space. This size matches the file set used in previous work [23], which concluded thread-per-connection servers match or exceed the performance of event-based servers. Our HTTP traces, though synthetic, accurately recreate the file classes, access patterns (i.e., a Zipf distribution), and number of requests issued per (HTTP 1.1) persistent connection that are used in the static portion of SPECweb99.

3.3 Tuning

Each server architecture has a number of static configuration parameters that can affect its performance with respect to different workloads. Tuning involves running experiments to measure the performance of each server as the tuning parameters are varied. In tuning the performance of each server for the workload used in this study, our methodology was to start by choosing a range of connections and kernel-level worker threads or processes and then to produce the matrix containing the cross product of these two ranges. In the case of tuning Knot with application-level caching a third dimension, cache size, was also included. The original ranges were chosen to be sufficiently wide that the extremes would result in poor performance and that the sweet spot would be covered somewhere within the original matrix. The results of runs using the original matrix were then analyzed and used to create a more fine-grained matrix, which was then used to better determine the combination of parameters that resulted in the best performance. In some cases further refinements were made to the fine-grained matrix and extra experiments were run to help provide insights into the behaviour of the different architectures. Each experiment requires between 2 and 8 minutes to run (depending on the rate at which requests were being issued) with 2 minutes idle time being used between experiments to ensure that all timed wait states would have a chance to clear. As a result, 30-75 minutes were required to produce a single line on a graph plotting multiple rates. Because of the time required to tune each server, our experiments in this paper are restricted to a single workload. In the future we hope to study alternative workloads.

Note that we only show a small subset of the experiments conducted during this study. They have been chosen to illustrate some of the issues that need to be considered when tuning these servers and to show the best performance tunings for each server.

3.4 Verification

Prior to running a full slate of experiments, a correctness test was conducted with each server to ensure that they were responding with the correct bytes for each request. In addition, it is important to verify that each server and each different configuration of the servers, successfully processes the given workload in a way that permits fair comparisons. Our SPECweb99-like workload uses a set of files with 36 different sizes, ranging from 102 to 921,600 bytes. Because some servers might obtain higher throughput or lower response times by not properly handling files of all sizes, we check if all file sizes are serviced equally. Client timeouts are permitted across all file sizes; however, the verification ensures that if timeouts occur, all file sizes timeout with approximately the same frequency (no single timeout percentage can be 2% larger than the mean). We also check that the maximum timeout percentage is $\leq$ 10%. An additional criteria is that no individual client experiences a disproportionate number of timeouts (i.e., timeout percentages for each file size are $\leq$ 5%). This check is similar in nature to that performed in SPECweb99 to verify that approximately the same quality of service is afforded to files of different sizes. Results are only included for experiments that pass verification for all request rates. In instances where verification did not pass, it was most often due to servers not being able to respond to requests for large files prior to the client timing out. This simulates a user getting frustrated while waiting for a page and stopping the request and browsers that have built-in timeout periods.

4. KNOT

In this section, the Capriccio threading library and the Knot web server are described, followed by a description of our modifications to Capriccio and Knot to support the `sendfile` system call. We then explore the impact of key performance parameters on three different versions of Knot: one using application-level caching (as in the original version) and two using different approaches to implementing `sendfile`. In the process, we demonstrate that both Knot and Capriccio are properly tuned for our workload. This tuning is a necessary precondition for a fair comparison of Knot, the µserver, and WatPipe in Section 7.

4.1 Capriccio Threading Library

Capriccio provides a scalable, cooperatively scheduled, user-level threading package for use with high concurrency servers. Previous work [23] has established that Capriccio scales to large numbers of threads with lower overheads than other Linux threading packages. As a result, Capriccio represents the state-of-the-art in Linux threading packages.

In Capriccio, multiple user-level threads are layered over a single kernel thread. By default, all sockets are set to non-blocking mode. Non-blocking socket I/O is performed by a scheduler thread executing on the same kernel thread as the user-level threads. Disk I/O is performed by handing off system calls that access the file system to auxiliary kernel-threads called worker threads. Capriccio carefully controls how system calls involving I/O are invoked by providing wrappers for most system calls that do socket or disk I/O.

For socket I/O, the user-level thread, via the system-call wrapper, attempts the system call in non-blocking mode. If the call completes successfully, the requesting thread continues. However, if the call returns `EWOULDBLOCK`, an I/O request structure is created and queued for processing and the socket descriptor is added to a `poll` interest set. The scheduler thread periodically invokes the `poll` system call to determine if the socket is readable or writable. When `poll` indicates that the socket is readable or writable, the system call is repeated and the requesting thread is then returned to the scheduler's ready list. All of our Knot experiments use the widely supported `poll` interface because `epoll` is only available in Linux systems and Capriccio's support for Linux's `epoll` interface is incomplete (the Capriccio paper reports that only the micro-benchmarks were run using `epoll` and that they experienced problems using `epoll` with Knot). Additionally, as shown in Section 7, the overheads incurred due to `poll` are small for well tuned servers and are in line with the `poll` or `select` overheads incurred by the other servers.

For disk I/O, an I/O request structure is created and queued for processing. Capriccio uses worker threads which periodically check for and perform the potentially blocking queued disk I/O requests. When a request is found, it is dequeued and the system call is performed by the worker thread on behalf of the user-level thread. As a result, calls block only the worker threads and the operating system will context switch from the blocked kernel thread to another kernel thread that is ready to run. When the disk operation completes, the worker thread is then scheduled by the kernel, the system call completes, and the requesting user-level thread is returned to the user-level thread scheduler's ready list.

4.2 Knot Server

The Knot server associates each HTTP connection with a separate Capriccio thread, creating a thread-per-connection server. Threads can be statically pre-forked, as part of the server's initialization, or dynamically forked when a new connection is accepted. Previous research [22, 5] reports that statically pre-forking threads results in better performance. As a result, pre-forked threads are used in our Knot experiments.

When threads are pre-forked, each thread executes a continuous loop that accepts a client connection, reads an HTTP request, and processes the request to completion before reading the next request from that client. Once all requests from a client are processed, the connection is closed and the loop is repeated. In Knot, the number of user-level threads places a limit on the number of simultaneous connections the server can accommodate. As a result, workloads that require large numbers of simultaneous connections require the server to scale to large numbers of threads. However, the need for a high degree of concurrency must be balanced against the resources consumed by additional threads. We explore this balance by tuning the number of threads used by Knot to match our workload.

Knot uses an application-level cache that stores HTTP headers and files in user-space buffers. An HTTP request causes an HTTP header to be created and data to be read into the cache buffers, if not already there. Then, the `write` system call is used to send these buffers to the client. However, the `write` call requires the buffer to be copied into the kernel's address space before the data can be transmitted to the client.

Intuitively, storing more files in the application-level cache reduces disk reads. However, a larger application-level file-cache can result in contention for physical memory. Thus, we explore the influence of application-level cache sizes on the performance of Knot.

4.3 Modifying Knot

Several changes were made to both Capriccio and Knot. First, Knot's cache was modified so it no longer stores the contents of files in application-level buffers. Instead, the cache stores only HTTP headers and open file descriptors. This change was made in preparation for subsequent changes that allow Knot to use the zero-copy `sendfile` system call. Second, the hash function used in the caching subsystem was changed (in all versions of Knot) to

a hash function taken from the µserver. This function is less sensitive to similarities in URIs present in our workloads, and the use of a common hash-function eliminates performance differences that might otherwise be caused by difference in the behaviour of the application's cache. Third, Capriccio and Knot were modified to support the `sendfile` system call. This step was taken for three reasons: 1) it allows Knot to take advantage of zero-copy socket I/O when writing replies, eliminating expensive copying between the application and the kernel, 2) it reduces the server's memory footprint by relying only on the file system to cache files, 3) it is viewed as part of the best-practices deployed by modern servers and allows for a fair comparison against the µserver and WatPipe, which can both use `sendfile` for sending data to clients. Since all servers used in this study can take advantage of zero-copy `sendfile` and leverage the file system cache, there are no performance differences due to data-copying, application-level cache misses, or memory consumption that can be caused by difference in application-level caching implementations.

Three different approaches to adding `sendfile` support to Capriccio were considered. In each case, a `sendfile` wrapper was created that is structurally identical to Capriccio's existing disk I/O wrappers, and each implementation required additional Capriccio changes (because `sendfile` could potentially block on socket I/O or disk I/O). As a result, Knot uses the same application-level code in each case. At a high-level, the three different approaches can be divided into two categories, based on whether or not the socket is in *non-blocking* or *blocking* mode.

For the *non-blocking sendfile* implementation, two implementations were tried. The first implementation leaves the socket in non-blocking mode. The application-level call to `sendfile` invokes a wrapper that adds the socket to the poll interest set, suspends the user-level thread and context switches to the scheduler thread. When a subsequent call to `poll` by the scheduler thread indicates that the socket is writable, the actual `sendfile` system call is performed by a worker thread. The worker thread only blocks if disk I/O is required. However, if the file's blocks are present in the file system cache, no disk I/O is required. Data is transmitted on the outgoing socket until either the socket blocks or the operation completes. In either case, the system call returns and the user-level thread is returned to the scheduler's list of ready threads. When the user-level thread restarts in the wrapper, the result from the worker thread is returned to the application, which either makes additional `sendfile` calls if all the bytes are not sent, retries the call, or fails with an error. Under some conditions, this version is susceptible to quite high polling overhead. Hence, all of the results reported use our second non-blocking implementation.

The second implementation leaves the socket in non-blocking mode. The application-level call to `sendfile` invokes a wrapper that adds the request to the worker-thread queue, suspends the user-level thread, and context switches to the scheduler thread. The actual `sendfile` system call is performed in Capriccio by a worker thread on behalf of the user-level thread. If the call successfully sends any portion of the data, the user-level thread is returned to the scheduler's ready queue. When the user-level thread restarts, the application examines the number of bytes sent, and decides if additional `sendfile` calls are needed. If the system call is unable to write data to the socket without blocking, `errno` is set to `EWOULDBLOCK`, and the worker thread blocks by calling `poll` on that socket, with a one second timeout. The timeout prevents the worker thread from being blocked for extended periods of time waiting for the socket to become ready. When the socket becomes writable or one second has elapsed, the worker thread unblocks. If the socket becomes writable before the timeout, another `sendfile` call is performed and the result of the `sendfile` is returned. If the `poll` call times out, `EWOULDBLOCK` is returned. In both cases, the user-level thread is returned to the scheduler's ready queue. When the user-level thread restarts, the application examines the result and either makes additional `sendfile` calls if necessary, retries the call, or fails with an error.

For the *blocking-sendfile* implementation, the application-level call to `sendfile` invokes a wrapper, which sets the socket to blocking mode, adds the request to the worker-thread queue, suspends the user-level thread, and context switches to the scheduler thread. The actual `sendfile` system call is performed by a worker thread. If the `sendfile` call blocks on either socket or disk I/O, the worker thread blocks until the call completes. As a result, only one `sendfile` call is required to write an entire file. Once the call completes, the associated user-level thread is returned to the scheduler's ready queue. When the user-level thread restarts, the application only needs to check for errors.

We modified Knot to make use of the new Capriccio `sendfile` wrapper for writing the file portion of HTTP replies and tuned the number of user-level threads and worker threads for each of the `sendfile` implementations.

4.4 Tuning Knot

To tune the three versions of the Knot server, called knot-c (application-level caching), knot-nb (non-blocking `sendfile`) and knot-b (blocking `sendfile`), an extensive set of experiments were run for each version varying the independent parameters: cache size for knot-c, number of threads (i.e., maximum simultaneous connections), and number of workers. In general, only a small subset of our experimental results are presented in order to illustrate some of the tradeoffs in tuning a thread-per-connection server like Knot.

Figure 1 shows how varying the cache-size parameter affects throughput for the application-level caching version of Knot (knot-c). A legend label, such as *knot-c-20K-100w-1000MB*, means 20,000 threads (20K), 100 workers (100w), and a 1000 MB cache (1000MB) were used. All graphs follow a similar labelling convention. Our tuning experiments found that 20,000 threads, 100 workers and a 1000 MB cache was one configuration that produced the best performance for knot-c, so we concentrate on how varying the cache size changes performance while maintaining the same number of threads and workers. The question of how the number of threads and workers affect performance is explored in other server configurations. The graph shows throughput is relatively insensitive to the size of the application-level cache in the range 100 to 1000 MB, with performance rising to a sustained level close to 1100 Mbps with a peak of 1085 Mbps for configuration *knot-c-20K-100w-1000MB*. Only when a very small cache size (e.g., 10 MB) is used does throughput begin to degrade. As shown in upcoming experiments, the performance of knot-c turns out to be relatively good even with considerable amounts of data copying required to move data between the application-level cache and the kernel.

Figure 2 shows how varying the number of user-level and worker threads affects throughput for the non-blocking `sendfile` version of Knot (knot-nb). With respect to the number of threads, the graph shows that too few threads hinders performance. With 10,000 (or fewer) threads and any number of workers, throughput never exceeded 930 Mbps. In this case, performance for knot-nb is limited because it is not supporting a sufficiently large number of simultaneous connections. As will be seen in subsequent experiments with all other servers, their performance is also limited to around 930 Mbps when the number of simultaneous connections is

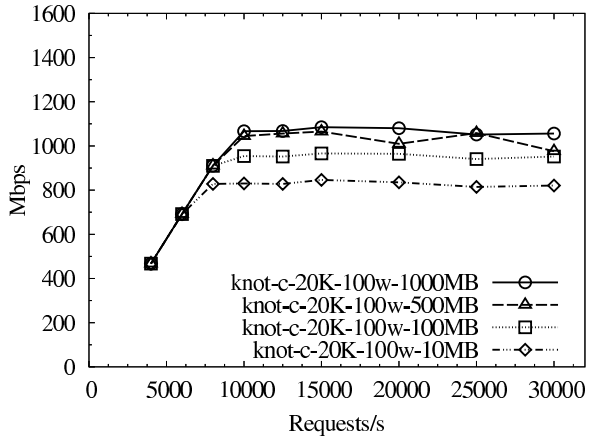

Figure 1: *Tuning Knot – application-level caching*

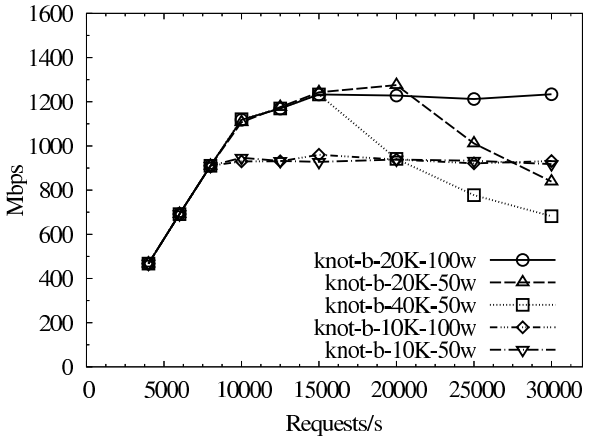

Figure 3: *Tuning Knot – blocking sendfile*

capped at 10,000.

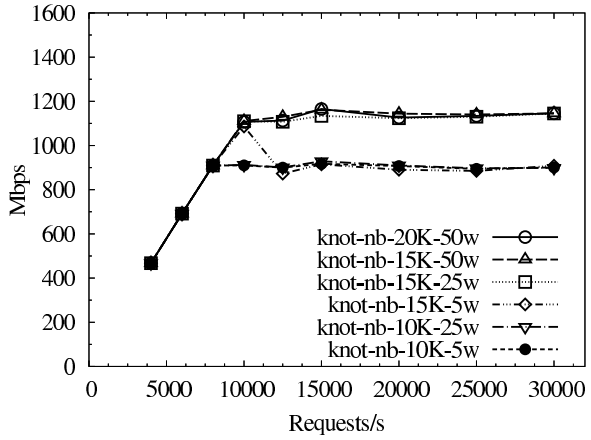

Figure 2: *Tuning Knot – non-blocking sendfile*

Increasing the number of threads to 15,000 while using 5 workers (*knot-nb-15K-5w*) results in an increase in peak performance at a request rate of 10,000 requests per second but performance drops as the request rate is increased. By increasing the number of workers to 25 performance improves substantially. No significant performance improvements were realized by increasing the number of threads or workers above 15,000 threads and 25 workers, indicating this server implementation has reached its maximum capacity for this workload (a sustained throughput of 1165 Mbps with a peak of 1200 Mbps).

Figure 3 shows how varying the number of user-level and worker threads affects throughput for the blocking `sendfile` version of Knot (knot-b). With respect to the number of threads, the graph shows that too few threads hinders performance. Again, with 10,000 (or less) threads and any number of workers, throughput is in most cases around 930 Mbps and never exceeds 960 Mbps. By increasing the maximum number of connections performance improves to a sustained level above 1200 Mbps with a peak of 1230 Mbps using *knot-b-20K-100w*.

For all Knot servers, it is crucial to match the number of threads and workers to achieve good performance. This effect is illustrated in knot-nb by a performance drop after the peak for *knot-nb-15K-5w* because there are insufficient worker threads to support the demands of the 15,000 user-level threads. The *knot-b-20K-50w* configuration has the same problem. Comparing the performance of configurations *knot-b-20K-50w* and *knot-b-40K-50w*, the performance degradation due to the lack of sufficient worker threads is even more pronounced in the *knot-b-40K-50w* case. A consequence of not having enough worker threads is that the poll set becomes large, significantly increasing the cost of a `poll` call. Knot-b is examined in detail to illustrate this point.

Statistics collected in the server show the number of calls to poll do not increase significantly as the request rates increase. However, the poll overheads for *knot-b-20K-50w* are 8%, 15% and 22% at request rates of 12,000, 25,000 and 30,000, respectively. The increase is even greater in the *knot-b-40K-50w* case, with poll overheads of 8%, 25% and 29% at request rates of 12,000, 25,000 and 30,000, respectively. The cause of this overhead is an increase in the average number of file descriptors (fds) in the poll set. In these two configurations the average number of fds is just under 100 for request rates of 15,000 requests per second. With 20,000 connections this increases to 243 and 432 and with 40,000 connections this increases to 459 and 609 at 25,000 and 30,000 requests per second, respectively. Because each file descriptor must be checked to see if there are events of interest available, the time spent in poll increases as the number of fds of interest increases. Increasing the number of workers from 50 to 100 is sufficient to support the number of threads and eliminates the dramatic decline in throughput as the request rate increases. With 20,000 threads and 100 workers (*knot-b-20K-100w*) a sustained throughput above 1200 Mbps is reached for the higher request rates. In this case, poll overhead is 8%, 6% and 5% at request rates of 12,000, 25,000 and 30,000, respectively.

5. THE µSERVER

The µserver can function as either a single process event-driven (SPED) server, a symmetric multiple process event-driven (SYMPED) server, or a shared symmetric multiple process event-driven (shared-SYMPED) server.

5.1 SYMPED Architecture

The µserver uses an event notification mechanism such as `select`, `poll`, or `epoll` to obtain events of interest from the operating system. These events typically indicate that a particular socket is readable or writable, or that there are pending connections on the listening socket. For each such event, the µserver invokes an event-handler that processes the event using non-blocking socket I/O. Once all events have been processed, the µserver retrieves a new batch of events and repeats the cycle.

In SYMPED mode, the µserver consists of multiple SPED processes. Each process is a fully functional web server that accepts new connections, reads HTTP requests, and writes HTTP replies. However, when one SPED process blocks due to disk I/O, the operating system context switches to another SPED process that is ready to run. This approach allows a high-performance SPED server to be used in environments where a single copy of the server blocks due to disk I/O. In our implementation, with the exception of sharing common listening sockets, all SPED processes are completely independent. As a result no coordination or synchronization is required among processes. Currently the number of SPED processes is specified as a command-line parameter, although we plan to implement dynamic tuning of this parameter in the near future.

The SYMPED model is similar to what has been previously described as the `N` -copy approach [25]. The `N` -copy model was used and evaluated as a means of enabling a SPED web server to leverage multiple CPUs in a multi-processor environment. The name is derived from the fact that on a host with `N` processors, `N` copies of a SPED web server are run. In the experiments conducted by Zeldovich et al. [25], they configured each copy to handle connections on a different port. Their experimental evaluation showed that this approach is highly effective in environments where communication is not required among the different copies of the web server. Utilizing the `N` -copy approach in production environments requires some method for efficiently multiplexing clients to different ports in a fashion that balances the load across all `N` copies.

Unlike the `N` -copy model, the SYMPED architecture allows all processes to share listening sockets by having the main server process call `listen` before creating (forking) multiple copies of the server. This permits the use of standard port numbers and obviates the need for port demultiplexing and load balancing. We believe that the SYMPED model is beneficial for both ensuring that progress can be made when one copy of the SPED server blocks due to disk I/O and for effectively utilizing multiple processors. Our results in this paper demonstrate that on a single CPU, with a workload that causes a single SPED server to block, a SYMPED server can fully and efficiently utilize the CPU.

In contrast with Knot (knot-c), the µserver is able to take advantage of zero-copy `sendfile`, and only caches HTTP headers and open file descriptors. As a result, there was no need for immediate modifications to the µserver code base.

5.2 Shared-SYMPED Architecture

While it was easy to convert our SPED server into a SYMPED server, this results in several processes each executing in their own address space. The advantage of this approach is that these processes execute independently of each other. The drawback is that each process maintains its own copy of the cache of open file descriptors and result headers, resulting in increased memory consumption and a possibly large number of open file descriptors. For example, with 25,000 open files in the cache and 200 SPED processes, the server requires 5,000,000 file descriptors to keep a cache of open files. Because this resource requirement could put a strain on the operating system when large numbers of processes are required, we implemented a shared SYMPED architecture in which each SYMPED server is augmented with shared-memory to store the shared cache of open file descriptors and result headers. These modifications were relatively straightforward using `mmap` to share application-level cache memory, `clone` to share a single open file table among processes and a futex [10] to provide mutually exclusive access to the shared cache. This approach significantly reduces the total number of required open file descriptors and also reduce the memory footprint of each µserver process.

5.3 Tuning the µserver

The main µserver parameters of interest for this workload are the number of SPED processes being run and the maximum number of simultaneous connections allowed per process (max-conn parameter). Both parameters were independently tuned. Additionally, we were interested to see how a version using blocking socket I/O calls would perform against the blocking-sendfile version of Knot. Figures 4 and 5 present results from several of the more interesting experiments.

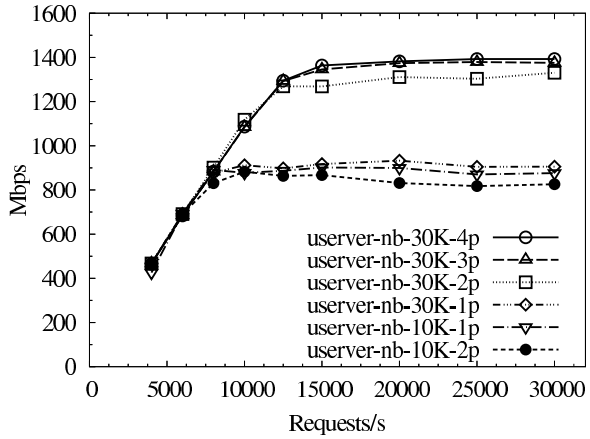

Figure 4: *Tuning the µserver – non-blocking sendfile*

Figure 4 shows how the µserver's throughput changes as the number of connections and SPED processes is increased. In these experiments, all socket I/O is non-blocking and the maximum connections permitted (max-conn value) is 10,000 or 30,000. Figure 4 also shows the throughputs obtained with 1, 2, 3, and 4 processes. For example, the line labelled *userver-nb-30K-3p* represents the µserver being run with a maximum of 30,000 concurrent connections across 3 server processes (i.e., 10,000 connections per process) and the line labelled *userver-nb-10K-2p* uses 10,000 connections across 2 processes (i.e., 5,000 connections per process). Other configurations are labelled similarly. In practice, the actual connection count may be lower than the upper bound set by the max-conn value. The results in Figure 4 again demonstrate the importance of supporting a sufficiently large number of concurrent connections. It also shows that a sufficient number of symmetric server processes must be used to ensure they are not all blocked waiting on I/O.

The better performing 30,000 connection cases are discussed first. The results from the *userver-nb-30K-1p* and *userver-nb-30K-2p* configurations show how the extra server process boosts peak throughput by 38% at 15,000 requests per second and by 47% at 30,000 requests per second. The key to this increase lies in the reduction of I/O waiting times. With a single process (*userver-nb-30K-1p*), under a load of 15,000 requests per second, the processor is idle for 33% of the time while the µserver is blocked waiting for I/O requests to complete. Because there is no other server process to run, this essentially robs the server of 33% of its execution time. However, adding a second process lowers CPU waiting time to 8%, which leads to the aforementioned increases in throughput.

A subsequent increase to the number of SPED processes continues to improve performance. With 3 server processes, the µserver spends just 5% of the CPU time waiting for I/O, and with 4 server processes waiting time is reduced to 4%. However, at this point adding more processes does not improve performance (experiments not shown). For this workload, relatively few processes are needed

to ensure the server is able to continue to make progress because many requests are serviced without requiring blocking disk I/O (they are found in the file system cache). Recall that the file set is not that large (3.2 GB) relative to the amount of memory in the system (2.0 GB) and because file references follow a Zipf distribution [21] most of the requests are for a small number of files.

The poorer performing 10,000 connections cases are included to illustrate that the small number of connections limits the throughput of the µserver just as is the case for knot-nb and knot-b (see Figures 2 and 3). It is also interesting to see that when comparing *userver-nb-10K-1p* and *userver-nb-10K-2p* the addition of one more process actually slightly degrades performance. In fact, continuing to add more processes (results not shown) continues to degrade performance due to significant increases in `poll` overhead. As the number of processes is increased, the number of connections managed by each process decreases. As a result for each call to `poll` fewer file descriptors are returned, which both decreases the amount of processing done between calls to `poll` and increases the number of calls to `poll`. The results shown in this figure again emphasize the fact that the number of connections and processes significantly affects performance and that they must be properly tuned.

Figure 5 shows the results of several interesting experiments with the µserver using blocking sockets for `sendfile` calls. Note, more processes are required to handle the same workload (75 to 200 compared with 3 or 4 in the non-blocking case) because calls to `sendfile` can now block due to file-system cache misses and because socket buffers are full.

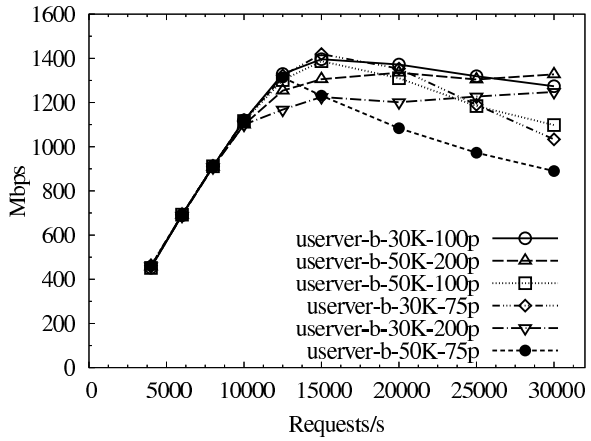

Figure 5: *Tuning the µserver – blocking sendfile*

Figure 5 shows the important interplay between the number of simultaneous connections and the number of processes. Comparing configurations *userver-b-30K-75p* and *userver-b-50K-75p* with 75 processes, performance is significantly better with 30,000 connections than with 50,000 connections. In this case, 50,000 connections is too many for the 75 processes to handle. Increasing the number of processes to 100 increases throughput for both 30,000 and 50,000 connections (*userver-b-30K-100p* and *userver-b-50K-100p*), which indicates 75 processes is insufficient in both cases. Increasing the number of processes in the 30,000 connections case to 200 actually hurts throughput (*userver-b-30K-200p*). In this case, there are now too many processes for the number of connections and the extra processes consume resources (e.g., memory) without providing additional benefits. However, increasing the number of processes to 200 in the 50,000 connections case provides quite good throughput (*userver-b-50K-200p*) with throughput being close to but slightly lower than the *userver-b-30K-100p* case.

Experiments conducted with the shared-SYMPED version of the µserver performed as well as the best performing SYMPED configurations. Due to differences in tuning parameters and the lack of memory pressure in the environment, the benefits of requiring less memory in the shared-SYMPED case are not seen. Furthermore, in this environment lock overhead and contention in the shared-SYMPED case are not an issue.

6. WATPIPE SERVER

A pipeline architecture transforms the steps for processing an HTTP request into a series of stages. These stages are typically self-contained and linked using a communication mechanism such as queues. Each stage is serviced by one or more threads, which can be part of per stage or shared thread pools. The most well known pipeline architecture is SEDA [24]. However, other types of pipeline architecture are possible.

WatPipe is our server based on a specific pipeline architecture. It is implemented in C++ and is built from the µserver source, so much of the code base is the same or similar; however, the components are restructured into a pipeline architecture. While SEDA is designed to allow for the creation of well-conditioned servers via dynamic resource controllers, WatPipe eliminates these controllers to simplify design while still achieving good performance. WatPipe uses a short pipeline with only a small number of threads in each stage. Next, it uses `select` to wait for events. Rather than using explicit event queues, each stage maintains its own read and/or write `fd_sets`. Periodically, the stages synchronize and merge their respective `fd_sets`. The idea is for each stage to perform a batch of work and then to synchronize with the next stage. Finally, Pthreads are used to create multiple threads within the same address space. WatPipe relies on Pthread's use of kernel threads, so each thread may block for system calls; hence, no wrapping of system calls is necessary. Like the µserver, WatPipe takes advantage of zero-copy `sendfile` and uses the same code as the µserver to cache HTTP reply-headers and open-file descriptors. As well, only a non-blocking `sendfile` version is implemented.

In contrast, SEDA and Haboob are implemented in Java. Haboob has a longer pipeline and utilizes dynamic resource controllers to perform admission control to overloaded stages. We believe WatPipe's careful batching of events and shortened pipeline should lead to fewer context switches. Specifically, the implementation consists of 4 stages. The first 3 stages have one thread each, simplifying these stages as there is no concurrency within a stage, and stage 4 has a variable number of threads. Synchronization and mutual exclusion is required when communicating between stages and when accessing global data (e.g., the open file descriptors cache). Stage 1 accepts connections and passes newly accepted connections to stage 2. Stage 2 uses `select` to determine which active connections can be read and then performs reads on these connections. Valid requests are passed to stage 3. Stage 3 uses `select` to determine which connections are available for writing. Once stage 3 determines the connections that can be written, the threads in stage 4 perform the actual writes. Stage 3 and stage 4 are synchronized so only one stage is active at a time. Because `sendfile` is non-blocking, an fd may cycle between stages 3 and 4 until all bytes are written. After all the data is written, the connection is passed back to stage 2 to handle the next request, if necessary. Having multiple threads performing the writing allows processing to continue even when a thread is blocked waiting for disk I/O to occur.

6.1 Tuning WatPipe

The main WatPipe parameters of interest for this workload are the number of writer threads in stage 4 and the maximum number of simultaneous connections allowed (max-conn parameter). Both parameters were independently tuned.

Figure 6 shows how WatPipe's throughput changes as the number of connections and writer threads are varied. The bottom two lines show the results obtained with a maximum of 10,000 concurrent connections, while varying the number of writer threads. The top three lines show the results obtained with a maximum of 20,000 concurrent connections, while varying the number of writer threads.

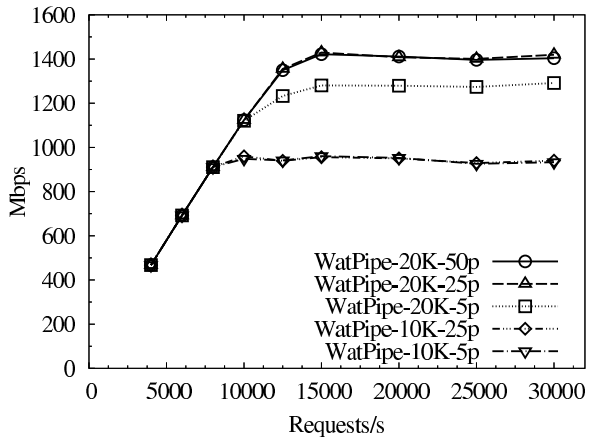

Figure 6: *Tuning WatPipe*

For this workload, 10,000 connections is insufficient. With 5 writer threads, increasing the number of connections from 10,000 to 20,000 boosts peak throughput by 33% at 15,000 requests per second and by 38% at 30,000 requests per second. However, increasing the number of connections above 20,000 did not result in further improvements (experiments not shown). Experiments were run for connections greater than 20,000; however, these experiments either did not improve performance or failed verification. Hence, adding further connections does not help as time is spent working on connections that eventually timeout.

The number of writer threads in stage 4 was varied to test the affect on performance. At 10,000 connections, increasing the number of threads has no affect because of the insufficient number of connections. With more connections, increasing the number of threads results in a performance improvement. For example, with a maximum of 20,000 connections, increasing the number of writer threads from 5 to 25 improves peak throughput by 12% at 15,000 requests per second and by 10% at 30,000 requests per second. The percentage of time the CPU was idle because processes were blocked waiting for disk I/O (as observed using vmstat) at 15,000 requests per second for 5 writer threads is 14%, and it deceases to 1% with 25 writers. This improvement is therefore attributed to the reduction in I/O wait times by having more writer threads that are ready to be run. However, further increasing the number of writer threads to 50 yielded the same results. Hence, adding more threads does not help because the I/O wait time has essentially been eliminated with 25 writers.

7. SERVER COMPARISON

In this section, a comparison is made among Knot, including both `sendfile` implementations, SYMPED and shared-SYMPED versions of the µserver, and WatPipe. In comparing these servers, special attention is paid to the tuning and configuration of each server. In all comparisons, the best tuning configuration for each server is used.

We have sought to eliminate the effects of confounding factors such as caching strategies, hashing functions, event mechanisms, or the use of zero-copy `sendfile`. As such, all the servers except for knot-c share similar caching strategies (caching open file descriptors and HTTP headers using essentially the same code), and all the servers use the same function for hashing URIs. In addition, all servers use the `poll` system call to obtain event notifications from the operating system, except WatPipe (which uses `select`). All servers except knot-c also use the Linux zero-copy `sendfile` system call for writing the file portion of the HTTP reply.

Our previous work [5] showed that a server's accept strategy (the method used to determine how aggressively a server accepts new connections) can influence performance by changing the rate at which the server accepts new connections. Because our HTTP workload has been designed to place high connection loads on the server, each server's accept strategy was chosen based on our prior experience. The µserver uses a strategy that repeatedly calls accept in non-blocking mode until the call fails (setting `errno` to `EWOULDBLOCK`). For Knot we use what is called the Knot-C configuration that favours the processing of existing connections over the accepting of new connections [22]. WatPipe uses a separate thread in the pipeline that is dedicated to accepting new connections. Therefore, we believe the differences in performance presented here are due primarily to specific architectural efficiencies, rather than tuning or configuration details.

Figure 7 presents the best performance tuning for each server architecture: caching Knot (knot-c), blocking Knot (knot-b), non-blocking Knot (knot-nb), non-blocking SYMPED µserver (userver-nb), blocking SYMPED µserver (userver-b), non-blocking shared-SYMPED µserver (userver-shared-nb), and WatPipe (WatPipe). In addition, a SPED server, which is the SYMPED server with one process, is shown as a baseline (*userver-nb-30K-1p*). The legend in Figure 7 is ordered from the best performing server at the top to the worst at the bottom.

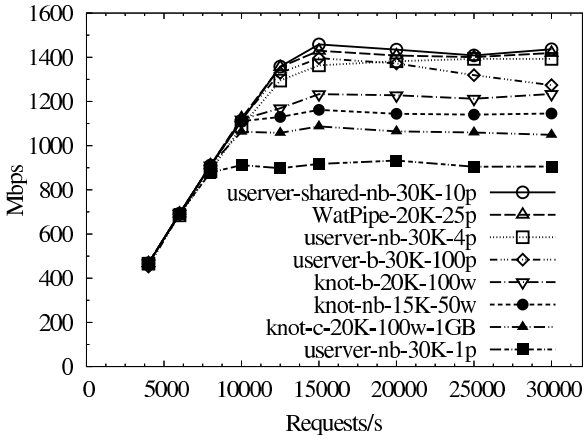

Figure 7: *Throughput of different architectures*

The top three servers have approximately equivalent performance. The fourth server, userver-b, has the same peak at 15,000 requests per second, but its performance deteriorates somewhat after saturation (e.g., there is a 12% drop in throughput when compared with userver-shared-nb at 30,000 requests). The next three servers are knot-b, knot-nb and knot-c, with knot-b having 13%

higher throughput than knot-c and 7% higher throughput than knot-nb at 15,000 requests per second. None of these servers achieved the peak obtained by the top four servers. The userver-shared-nb server provides 18% higher throughput than the best of the Knot servers (knot-b) at 15,000 requests per second and 34% higher throughput than knot-c at the same request rate. The worst performing server is the baseline SPED server (*userver-nb-30K-1p*), which shows how a single process event-driven server compares to the more complex servers. It can also be used to gauge the intensity of disk activity caused by the workload. Under this workload, the SPED server is within 37% of the peak despite spending 33% of its time blocked waiting for I/O.

In order to understand the performance differences among the servers, in depth profiling was performed for some of the best server configurations. In particular, each server was subjected to a load of 12,000 requests per second, while running OProfile and vmstat. This rate captures peak performance with OProfile overheads, yet ensures that all runs pass verification. OProfile performs call analysis via sampling. Statistics are also gathered directly from each server. The resulting data are summarized in Table 1, which is divided into four sections.

The first section lists the architecture, configuration, reply rate, and throughput in megabits per second for each server. Under the architecture row (labelled "Arch") the entry "T/Conn" refers to a thread-per-connection architecture and "s-symped" refers to a shared-SYMPED architecture. The second section provides a breakdown of CPU utilization as reported by the OProfile system profiler. OProfile produces a listing of kernel and application functions and the percentage of CPU time spent in each function. These functions are divided among Linux kernel (vmlinux), Ethernet driver (e1000), application (user space), and C library (libc). All remaining functions fall into the "other" category. This category mostly represents OProfile execution. Within these categories, each function is then assigned to one of the listed sub-categories. Categorization is automated by generating ctags files to define the members of each sub-category. In the user-space category, threading overhead denotes time spent in the threading library (Capriccio for Knot and Pthreads for WatPipe) executing code related to scheduling, context-switching, and synchronization of user-level threads. It also includes communication and synchronization between user and kernel threads for Knot. The event overhead refers to the server CPU time spent managing event interest-sets, processing event notifications from the operating system, and invoking appropriate event handlers for each retrieved event. The application sub-category includes the time not spent in thread and event overhead. The third section presents data gathered by vmstat during an experiment. The vmstat utility periodically samples the system state and reports data at a user-configured interval, which is set to five seconds in our experiments. A table entry is the average of the sampled values during an experiment. The row labelled "file system cache" gives the average size of the Linux file-system cache. The row labelled "ctx-sw/sec" gives the average number of context switches per second performed by the kernel. The last section contains the number of user-level context switches gathered directly from Capriccio. For each server, we discuss only the values where there is a significant difference among the servers.

Under this workload, Knot requires a large number of user-level threads ($\geq$ 10,000) to achieve good performance. As a result, all the Knot servers have an additional user-level threading overhead (from 7% to 11%), including user-level thread management, synchronization, and context switches. This overhead reduces the time available for servicing requests. As well, all Knot servers spend more time (2 to 4% more) in the application than the other servers.

The OProfile data for the knot-c configuration reveals large overheads due to kernel data copying (17.73%) and user-level threading (6.84%). Together these overheads account for 24.57% of CPU time, reducing the time available for servicing requests. The high data-copying overhead underscores the need for reducing data copying between the application and the kernel. Our modifications to Capriccio and Knot to utilize `sendfile` virtually eliminate data-copying overhead for knot-nb and knot-b. Finally, knot-c has lower network and e1000 overhead than all the other servers because of its reduced throughput.

The OProfile data for both knot-nb and knot-b show reduced data-copying overhead compared with knot-c, without any increase in the polling overhead; hence the `sendfile` modification does produce an increase in throughput. Note the increase in kernel context-switches in the knot-b case, when compared with the knot-nb case. In the knot-b case each of the many threads performs a blocking `sendfile` call resulting in a worker process potentially blocking. However, the context switching resulting from these operations seems to produce only a small increase in the scheduling overhead (1.64% versus 0.81%).

The remaining four servers all achieve higher throughput than Knot, independent of the their specific architecture. All of the servers in this group have lower user-space time than the Knot servers, mainly because they have little or no user-level threading overhead. Instead, a small amount of event overhead appears in the last four servers. Both userver-shared-nb and WatPipe have a small amount of threading overhead because each requires some synchronization to access shared data. Unfortunately, some of the `futex` routines used by userver-shared-nb are inlined, and hence, do not appear in the threading overhead; instead, these overheads appear in "application" time.

The userver-shared-nb ("s-symped") OProfile data does not show any anomalies. Performance of this version is equal to the best of all architectures and is reflected in the OProfile data.

The userver-nb OProfile data shows that polling overhead is twice as large as the userver-b overhead. This difference appears to be correlated to the larger size of the poll set required for userver-nb when compared with userver-b. When blocking socket I/O is used, `sendfile` only needs to be called once, and as a result, the socket being used does not need to be in the interest set, which can significantly reduce the size of the interest set. When comparing the architecture of the non-blocking µservers with their blocking counterparts, the non-blocking versions have the advantage of requiring fewer processes (e.g., 10 or 4 processes versus 100) resulting in a smaller memory footprint, which makes more memory available for the file system cache. Specifically, the vmstat data reported in Table 1 shows that with userver-nb the size of the file system cache is 1,803,570 megabytes versus 1,548,183 megabytes with userver-b. This advantage appears to have few or no implications under this workload in our environment, however it may be an important issue in a more memory constrained environment. As well, non-blocking versions incur less kernel context switching overhead.

The WatPipe OProfile data does not show any anomalies. Despite having a radically different architecture from the SYMPED servers, WatPipe has performance that is similar across all measured categories. However, while WatPipe does use non-blocking `sendfile`, it requires 25 writer threads to achieve the same performance as the non-blocking 10 processor µserver. We believe this results from a non-uniform distribution of requests across writer threads in the WatPipe implementation.

We also note that both userver-b and WatPipe have a medium number of kernel context switches. However, as is the case with the Knot servers, these context switches do not appear to have an

Server Arch Write Sockets	**Knot-cache T/Conn non-block**	**Knot T/Conn non-block**	**Knot T/Conn block**	**userver s-symped non-block**	**userver symped non-block**	**userver symped block**	**WatPipe SEDA-like non-block**
Max Conns Workers/Processes Other Config	**20K 100w 1 GB cache**	**15K 50w**	**20K 100w**	**30K 10p**	**30K 4p**	**30K 100p**	**20K 25p**
Reply rate Tput (Mbps)	8,394 1,001	8,852 1,055	9,238 1,102	10,712 1,280	10,122 1,207	10,201 1,217	10,691 1,276
OPROFILE DATA							
vmlinux total	**68.68**	**62.65**	**63.42**	**67.32**	**69.40**	**68.38**	**67.81**
networking	21.72	28.59	27.74	30.94	28.81	29.95	33.26
memory-mgmt	6.90	7.48	6.94	7.65	6.95	8.32	9.00
file system	5.17	7.42	7.55	8.22	9.32	8.18	6.65
kernel+arch	4.72	5.15	5.47	6.11	5.84	7.74	6.21
poll overhead	6.88	7.82	7.91	8.29	13.06	6.16	6.52
data copying	17.73	0.71	0.77	1.04	1.01	1.04	1.00
sched overhead	0.86	0.81	1.64	0.15	0.06	1.06	0.29
others	4.70	4.67	5.40	4.92	4.35	5.93	4.88
e1000 total	**13.85**	**16.42**	**15.77**	**17.80**	**16.74**	**16.77**	**19.17**
user-space total	**15.17**	**18.56**	**18.38**	**10.22**	**9.54**	**8.42**	**9.24**
thread overhead	6.84	10.52	10.20	0.01	0.00	0.00	2.29
event overhead	0.00	0.00	0.00	5.09	5.06	3.60	3.34
application	8.33	8.04	8.18	5.12	4.48	4.82	3.61
libc total	**0.03**	**0.03**	**0.03**	**2.60**	**2.47**	**4.15**	**1.65**
other total	**2.27**	**2.34**	**2.40**	**2.06**	**1.85**	**2.28**	**2.13**
VMSTAT DATA							
file system cache	750,957	1,773,020	1,748,075	1,817,925	1,803,570	1,548,183	1,777,199
ctx-sw/sec (kernel)	1,947	5,320	8,928	330	107	4,394	1,148
SERVER STATS							
ctx-sw/sec (user)	12,345	20,836	19,158	0	0	0	0

Table 1: ***Server performance statistics gathered under a load of 12,000 requests per second***

effect on throughput.

The OProfile data shows that one configuration (*userver-symped-nb-30K-4p*) incurs noticeably higher `poll` overheads than the other server configurations. Preliminary experiments with `epoll` yielded throughput that is as good as but no better than with `poll`, even though overheads directly attributable to `epoll` were noticeably reduced. Although previous experiments have also observed that `epoll` does not necessarily provide benefits when compared with `poll` [11], it would be interesting to examine whether or not `epoll` could provide benefits to the servers used in this study.

8. RESPONSE TIMES

For completeness, response time data was collected for each server configuration. In particular, we measured the average time taken for a client to establish a connection, send an HTTP request, and receive the corresponding reply. For HTTP 1.1 connections with multiple requests, the connection time is divided among the replies received on that connection. Figure 8 shows average response times for all of the server configurations shown in Figure 7.

This graph shows that each server architecture performs reasonably with respect to mean response times. Except for *userver-nb-30K-4p*, response times are lower at lower request rates and increase noticeably when the server reaches its saturation point (between 10,000 and 15,000 requests per second).

In the three cases where mean response times are higher

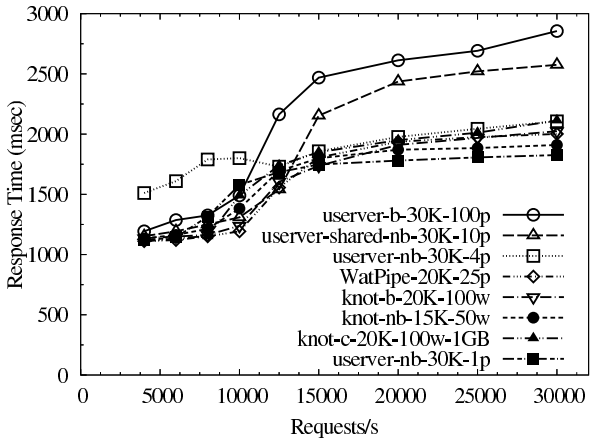

Figure 8: *Response times of different architectures*

(*userver-b-30K-100p*, *userver-shared-nb-30K-10p*, and *userver-nb-30K-4p*), it is due to the larger number of simultaneous connections. Each of these permits up to 30,000 connections and multiplexing across larger numbers of connections increases response time relative to multiplexing across a smaller number of connection. Reducing the maximum number of connections to 20,000 reduces response times to those of the other server configurations; however, this does slightly reduce throughput. With further tuning,

we might be able to obtain both high throughput and low response times (e.g., somewhere between 20,000 and 30,000 connections).

Although the *userver-nb-30K-1p* permits a maximum of 30,000 connections, at 30,000 requests per second, a maximum of only 2,615 connections are active at one time. This is due to the single process blocking on disk I/O and not being able to accept and process connections at the same rate as other server configurations. The result is lower throughput and slightly lower response times at higher request rates than the other configurations.

9. DISCUSSION

Our findings are substantially different from the conclusions reached in the comparison of the thread-per-connection Knot and the SEDA-based Haboob servers [22, 23]. One of the main differences is that we use efficient, scalable, event-driven and pipeline-based servers in our study. The µserver is implemented in C and features an event-driven core that when using the SYMPED or shared-SYMPED architecture scales to large numbers of concurrent connections. WatPipe is implemented in C++ and its simple hybrid design is able to keep both event and threading overheads low while being able to service large numbers of connections. In contrast, Haboob is implemented in Java and uses thread pools with complex resource controllers to process SEDA events.

The workload used in this paper is also substantially different from that used in the previous study. Our workload uses the httperf workload generator [16] to model a partially open system [20] and to force the server to handle overload conditions [2]. In addition, our HTTP workload models user think-times, as well as browser and network delays. The above factors create a workload that requires the server to efficiently scale to large numbers of concurrent connections.

We believe that there may be workloads for which thread-per-connection servers may provide performance equal to that offered by the other server architectures examined in this paper. An example is a workload comprised of short lived connections (i.e., each connection requests a small number of files with little or no delay between requests), where the requested files are serviced from the operating system file cache. This scenario does not require support for a large number of simultaneous connections, permitting a thread-per-connection server to execute with a relatively small number of threads, thus limiting overheads incurred when multiplexing between connections. However, on the workload used in our experiments, which requires a server to scale to large numbers of connections, there is a noticeable gap in performance. We also expect that this gap may increase if the workload forces the server to support an even larger number of simultaneous connections.

While the pros and cons of different server architectures have been covered extensively [15, 17, 12, 18, 8, 1, 25, 22, 9, 7], we believe that insufficient attention has been devoted to the fact that a user-level thread library like Capriccio is built on an event-driven foundation. As a result, the user-level thread-library incurs overheads that are most often only associated with event-driven applications. The threading layer then adds overheads for context-switching, scheduling, and synchronization. For file system I/O, the use of kernel worker threads to perform blocking operations adds further overheads for communication and synchronization between user-level threads and worker threads. In thread-per-connection servers that require lots of threads to support a large number of simultaneous connections these overheads can be significant. Our profiling data demonstrates that Capriccio's threading overheads consume 6% – 10% of available CPU time, thus hampering Knot's performance. The results obtained using the µserver and WatPipe show that architectures that do not require one thread for each connection do not incur such overheads, and as a result, provide better throughput.

We believe the performance of Knot and Capriccio can be further improved and it will be interesting to see if the current efficiencies in Capriccio can be maintained in an environment that supports multiprocessors or if such support necessarily introduces new inefficiencies. However, we also believe that as long as threading libraries rely on an underlying event-driven layer, the overhead incurred due to managing both events and a large number of threads will make it very difficult for thread-per-connection servers to match (let alone exceed) the performance of well implemented event-driven and pipeline-based servers under workloads that force servers to scale to a large number of concurrent connections.

10. CONCLUSIONS

This paper presents a performance-oriented comparison of event-driven, thread-per-connection, and hybrid pipelined server architectures. The µserver represents an event-driven design, the Knot server represents a thread-per-connection design, and WatPipe represents a pipeline-based approach. The µserver and Knot (with the Capriccio thread library) each represent the state-of-the-art in terms of performance in their respective categories. Because the previously best performing implementation of a pipeline-based architecture, Haboob, has not performed well in recent studies [23, 6, 19], we implement a simplified pipeline-based server in C++ based on the µserver.

We modify Capriccio and Knot to utilize zero-copy `sendfile` calls in Linux, implementing different approaches and comparing their performance with the original version that does not use `sendfile` but instead relies on an application-level cache implemented in Knot. We observe that the approach that uses non-blocking `sendfile` calls provides about 7 – 9% better throughput than the original version which uses an application-level cache (for request rates in the range of 12,500 to 30,000 requests per second). The approach that uses blocking `sendfile` calls in Capriccio produces 11 – 18% higher throughput than the original version of Knot that uses an application-level cache over the same range of rates. Interestingly, the performance of the non-blocking `sendfile` version is lower than that of the version that uses blocking `sendfile` calls.

We demonstrate the importance of properly tuning each server. For the workload used in this study, we show that a proper combination of the number of connections and kernel-level worker threads (or processes) is required. When comparing the experimental results obtained using the best tuning for each server the experimental results demonstrate that the event-driven µserver and the pipeline-based WatPipe achieve up to 18% higher throughput than the best implementation and tuning of Knot. Our detailed analysis reveals that Knot's throughput is hampered by overheads in Capriccio.

In the future, we plan to carry out similar studies with workloads that require both less and more disk activity to better understand how the performance of these architectures might be influenced. Ideally, this work could be done after or in conjunction with work that enables servers to automatically and dynamically tune themselves to efficiently execute the offered workload. We also hope to compare these architectures to other approaches such as Lazy Asynchronous I/O [9], which utilizes scheduler activations to improve the performance of event-driven servers under I/O intensive workloads. Finally, we intend to explore the performance of various server architectures using workloads containing dynamic content, and on multi-core, multi-processor hardware.

11. ACKNOWLEDGEMENTS

Funding for this project was provided by Hewlett-Packard, Intel, Gelato, the Natural Sciences and Engineering Research Council of Canada, and the Ontario Research and Development Challenge Fund. We thank Alan Cox and the anonymous reviewers for helpful comments and suggestions on earlier versions of this paper.

12. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[2] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey CA, December 1997.

[3] G. Banga, J. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

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

[5] T. Brecht, D. Pariag, and L. Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[6] B. Burns, K. Grimaldi, A. Kostadinov, E. Berger, and M. Corner. Flux: A language for programming high-performance servers. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 129–142, 2006.

[7] R. Cunningham and E. Kohler. Making events less slippery with eel. In *10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[8] F. Dabek, N. Zeldovich, M. F. Kaashoek, D. Mazires, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 186–189, September 2002.

[9] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[10] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast user-level locking in linux. In *Ottawa Linux Symposium*, June 2002.

[11] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the 6th Annual Ottawa Linux Symposium*, July 2004.

[12] J. Hu, I. Pyarali, and D. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*. IEEE, November 1997.

[13] H. Jamjoom and K. G. Shin. Persistent dropping: An efficient control of traffic aggregates. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

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

[15] H. Lauer and R. Needham. On the duality of operating systems structures. In *Proceedings of the 2nd International Symposium on Operating Systems, IRIA*, October 1978.

[16] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *The First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998.

[17] J. Ousterhout. Why threads are a bad idea (for most purposes), January 1996. Presentation given at the 1996 USENIX Annual Technical Conference.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[19] K. Park and V. S. Pai. Connection conditioning: Architecture-independent support for simple, robust servers. In *Network Systems Design and Implementation*, 2006.

[20] B. Schroeder, A. Wierman, and M. Harchol-Balter. Closed versus open system models: a cautionary tale. In *Network System Design and Implementation*, 2006.

[21] Standard Performance Evaluation Corporation. *SPECWeb99 Benchmark*, 1999. http://www.specbench.org/osg/web99.

[22] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea for high-concurrency servers. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[23] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[24] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, Banff, Oct. 2001.

[25] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX 2003 Annual Technical Conference*, June 2003.