Performance Comparison of Uniprocessor and Multiprocessor Web Server Architectures

by

Ashif S. Harji

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

Abstract

This thesis examines web-server architectures for static workloads on both uniprocessor and multiprocessor systems to determine the key factors affecting their performance. The architectures examined are event-driven (*μ*server) and pipeline (WatPipe). As well, a thread-per-connection (Knot) architecture is examined for the uniprocessor system. Various workloads are tested to determine their effect on the performance of the servers. Significant effort is made to ensure a fair comparison among the servers. For example, all the servers are implemented in C or C++, and support sendfile and edge-triggered epoll. The existing servers, Knot and *μ*server, are extended as necessary, and the new pipeline-server, WatPipe, is implemented using *μ*server as its initial code base. Each web server is also tuned to determine its best configuration for a specific workload, which is shown to be critical to achieve best server performance. Finally, the server experiments are verified to ensure each is performing within reasonable standards.

The performance of the various architectures is examined on a uniprocessor system. Three workloads are examined: no disk-I/O, moderate disk-I/O and heavy disk-I/O. These three workloads highlight the differences among the architectures. As expected, the experiments show the amount of disk I/O is the most significant factor in determining throughput, and once there is memory pressure, the memory footprint of the server is the crucial performance factor. The peak throughput differs by only 9–13% among the best servers of each architecture across the various workloads. Furthermore, the appropriate configuration parameters for best performance varied based on workload, and no single server performed the best for all workloads. The results show the event-driven and pipeline servers have equivalent throughput when there is moderate or no disk-I/O. The only difference is during the heavy disk-I/O experiments where WatPipe's smaller memory footprint for its blocking server gave it a performance advantage. The Knot server has 9% lower throughput for no disk-I/O and moderate disk-I/O and 13% lower for heavy disk-I/O, showing the extra overheads incurred by thread-per-connection servers, but still having performance close to the other server architectures. An unexpected result is that blocking sockets with sendfile outperforms non-blocking sockets with sendfile when there is heavy disk-I/O because of more efficient disk access.

Next, the performance of the various architectures is examined on a multiprocessor system. Knot is excluded from the experiments as its underlying thread library, Capriccio, only supports uniprocessor execution. For these experiments, it is shown that partitioning the system so that server processes, subnets and requests are handled by the same CPU is necessary to achieve high throughput. Both N-copy and new hybrid versions of the uniprocessor servers, extended to support partitioning, are tested. While the N-copy servers performed the best, new hybrid versions of the servers also performed well. These hybrid servers have throughput within 2% of the N-copy servers but offer benefits over N-copy such as a smaller memory footprint and a shared address-space. For multiprocessor systems, it is shown that once the system becomes disk bound, the throughput of the servers is drastically reduced. To maximize performance on a multiprocessor, high disk throughput and lots of memory are essential.

Acknowledgements

I would like to thank Peter Buhr, my supervisor, mentor and friend. You have always given me the trust and respect of a colleague, but also tremendous support and guidance. As a result, I have had a most unusual graduate experience. I have enjoyed the wide variety of projects we have worked on together, and I will miss our close collaboration.

I also thank the other members of my committee: Tim Brecht, Sebastian Fischmeister, Martin Karsten and Matt Welsh. Your feedback has improved this thesis and will help in my future research.

During my time in the Programming Languages lab, it has always been more than a place to work. The lab has undergone many changes over the years, but one constant is the interesting people who choose to work there. I would like to acknowledge all the members of PLG. You are a big part of the reason I stayed for as long as I did. In particular, I would like to thank my good friend Roy Krischer for his help and advice, and for his love of good food and movies. I would also like to thank Jiongxiong Chen, Maheedhar Kolla, Brad Lushman, Richard Bilson, Ayelet Wasik, Justyna Gidzinski, Stefan Büttcher, Mona Mojdeh, Josh Lessard, Rodolfo Esteves, Azin Ashkan, Ghulam Lashari, Jun Chen, Davinci Logan Yonge-Mallo, Kelly Itakura, Ben Korvemaker, John Akinyemi, Tom Lynam, Egidio Terra, Nomair Naeem, Jason Selby, Dorota Zak and Krszytof Borowski. It has been a privilege sharing a lab with all of you. As well, thank you to Lauri Brown, Stuart Pollock, Robert Warren, Elad Lahav, David Pariag and Mark Groves.

I would also like to thank the people I have had the pleasure of working with in my capacity as lab manager. Especially, Mike Patterson, Wendy Rush, Lawrence Folland and Dave Gawley.

To my parents and my brother, thank you for your tremendous support and patience.

To my parents

Contents

List of Tables

List of Figures

List of Abbreviations

AIO	Asynchronous I/O
AMPED	Asymmetric Multi-Process Event-Driven
API	Application Programming Interface
Gbps	Gigabits per second
IPC	Inter-Process Communication
IRQ	Interrupt Request
LRU	Least Recently Used
Mbps	Megabits per second
MP	Multiprocess
MT	Multi-threaded
SEDA	Staged Event-Driven Architecture
shared-SYMPED	Shared SYmmetric Multi-Process Event-Driven
SPED	Single Process Event-Driven
SYMPED	SYmmetric Multi-Process Event-Driven

Chapter 1

Introduction

One of the biggest problems for many Internet companies is scalability. With social networking and cloud computing growing in popularity, not only is more user-generated content moving online, ever larger online user-communities are placing increased demands on popular web-sites. For example, Facebook[1] serves over 600,000 unique images a second. Delivering this content to a growing user base presents many scalability problems. While additional hardware is necessary to handle the increased loads, correspondingly advanced servers are required to utilize this hardware and manage the increased loads. These servers must be able to handle high throughput and support a large number of concurrent connections. Fundamentally, web servers are a key component through which much Internet traffic flows. Another recent change is the shift to parallel processors (multi-threaded, multi-core, multiprocessor) even on low-end commodity hardware. This hardware has the potential to reduce the number of server machines necessary for large commercial sites, reducing power footprints and maintenance costs. However, this shift is forcing applications to become multi-threaded to take advantage of the parallel hardware. Little research has focused on examining web-server architectures for static content on multiprocessors. The assumption is existing server architectures that incorporate threading should scale and perform similarly on multiprocessors. However, previous work has shown this assumption to be false [62].

In order to achieve high performance, it is reasonable to use specialized, highly-tuned servers for various types of traffic. Specifically, a reasonable design is to use a separate server to handle static content [29, 31], possibly off-site, for example, at Amazon S3[2] or Akamai[3]. Even for general web-servers, efficiently handling static content frees up resources for other types of traffic. Static content is an important aspect of web traffic, in fact, much user-generated content is static. This thesis examines various

[1]http://www.facebook.com
[2]http://aws.amazon.com/s3/
[3]http://www.akamai.com

web-server architectures for static workloads on both uniprocessor and multiprocessor systems. The goal of the thesis is to identify the key factors affecting the performance of web servers.

Much research has focused on different architectures for web servers serving static content [17, 44, 57, 59, 61]. This work has led to a number of improvements in operating systems and web-server implementations; e.g., zero-copy transfer and user-level thread libraries that scale to thousands of threads. While comparisons have been performed as various improvements were developed, a fair comparison of the different architectures based on the current state-of-the-art for web servers, across a number of workloads, has not been performed. Therefore, the first objective of this thesis is to undertake such a comparison. The comparison begins by analysing the performance of uniprocessor servers, which can then be used as a baseline for examining the performance of multiprocessor architectures. The second objective of this thesis is to extend the uniprocessor servers to perform well on a multiprocessor and to compare their performance across two different workloads. Given the uniprocessor servers as a baseline, the approach of running multiple copies of a uniprocessor server [62] is the benchmark used to evaluate the performance of the extended servers.

1.1 Contributions

The contributions of this thesis are:

- A new pipeline server, WatPipe, is implemented. Its performance is comparable to the other servers examined for in-memory workloads and it has better performance for disk-bound workloads. As well, an existing thread library, Capriccio, is extended with a new non-blocking sendfile implementation.
- WatPipe and shared-SYMPED, an event-driven architecture, are extended for multiprocessor execution, including support for partitioning of processes/kernel-threads, subnets and CPUs. These extensions allow versions of these server to achieve performance comparable to N-copy on multiprocessors.
- Significant effort has been undertaken to make the servers as consistent as possible to ensure the comparison among the servers is fair. Contrary to previous work, it is shown that performance differences between state-of-the-art implementations of the various server-architectures is small given a level playing-field. In fact, the experiments in the thesis show architecture is most important as it relates to the memory footprint of the servers, when there is memory pressure.
- An important result in the thesis is that non-blocking sendfile is better when there is no disk I/O or moderate disk I/O, and blocking sendfile is better when the server is disk bound. Once the workload

is disk bound, the blocking version of a server performs better than the corresponding non-blocking server despite having a larger memory footprint because of better disk efficiency due to different file-access patterns. The exception is the blocking shared-SYMPED servers in the multiprocessor experiments, where their memory footprint is too large, negating the benefit.

- Through extensive experiments across a range of parameters, it is shown that proper tuning is critical to achieve best server performance and that no single tuning achieves the best performance for all workloads.

- Insight is provided into the work behind the thesis. Lessons learned are presented with respect to implementing, debugging and performing a large number of performance experiments on web servers.

1.2 Thesis Outline

The thesis is organized as follows. Chapter 2 covers the background material for the thesis and the related work. Chapter 3 compares the performance of several web-server architectures on a uniprocessor across three workloads, from in-memory to disk bound. The architectures tested include thread-per-connection, SYMPED, shared-SYMPED and pipeline. Each server is run with blocking and non-blocking sendfile and a version of the thread-per-connection server, using write with an application file-cache, is also tested. The performance of the servers are also profiled and analysed to understand their differences. These experiments establish a baseline for multiprocessor experiments. Chapter 4 compares the performance of several web-server architectures on a multiprocessor across two workloads, in-memory and disk bound. The architectures tested include SYMPED, shared-SYMPED and WatPipe. Both an N-copy version of the server and a version of the server extended for execution on a multiprocessor are tested. Again, the servers are run with blocking and non-blocking sendfile and then profiled and analysed to understand the performance differences. Chapter 5 relates some of the lessons learned based on the experience of implementing, debugging and performing a large number of experiments on the web servers in the thesis. Chapter 6 contains the conclusion and future work.

Chapter 2

Background and Related Work

A variety of architectures for building web servers have been proposed and implemented. These servers have become increasingly sophisticated to deal with a large number of concurrent connections while achieving high throughput. In addition to evolving server architectures, operating-system facilities have been examined and extensions proposed to improve scalability, efficiency and ease of programming. This chapter examines this work.

2.1 Handling an HTTP Request

Before looking at complex server-architectures, it is useful to understand how a server handles an HTTP request. Processing a typical static HTTP-request by a server consists of several steps, see Figure 2.1. The server begins by accepting a connection request from a client. Once a connection is established, an HTTP request is read from a socket. For a static request, the server searches for the requested file and a response is sent to the client. Assuming the request is successful, as part of this response the file is read from disk and sent over the network to the client. If the connection is persistent, the server reads a new request from the client. If there are no new requests, the connection is closed and a new connection is accepted. This description is simplified as it does not consider file caches, the potential need to send the file in blocks, etc.

Handling a single request can involve a number of delays, such as transmission latencies and bandwidth limitations. Given these delays, the time to handle a typical request can be measured in seconds [9]. Therefore, a server needs to handle many requests simultaneously. For a high-performance server to process thousands of requests a second, the server must be able to efficiently handle thousands of simultaneous connections. The web-server architectures examined in this thesis each have a different approach

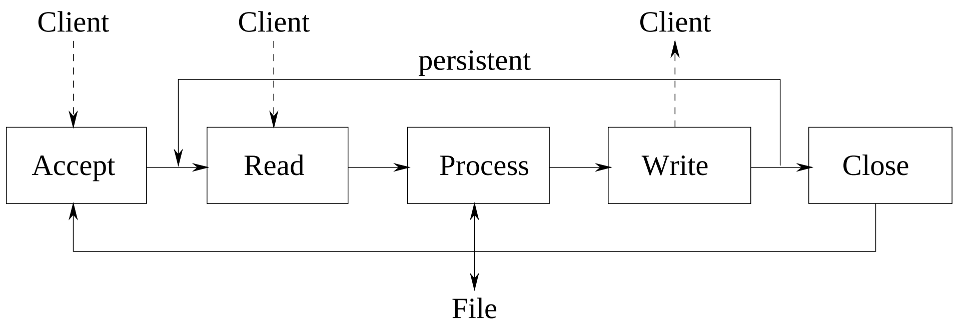


Figure 2.1: Server-side processing of an HTTP connection

to dealing with a large number of simultaneous connections.

2.2 Server Architectures

There are several different server architectures appropriate for high-performance web-servers. As well, there are a number of criteria upon which to evaluate the various server architectures. For this thesis, the most important criteria is throughput, both at peak and after saturation. Throughput is selected because it is a common performance metric used in a large body of previous work, allowing the results in the thesis to be compared to this work. A well-designed server should degrade gracefully after saturation, maintaining high throughput. As a consequence of the workloads used in this thesis, a server must support a large number of connections to achieve high throughput. Additional criteria affecting throughput include scalability, latency, memory footprint and contention, depending on the workload being tested. Finally, while ease of programming and debugging are important, they are not considered in this thesis because all the servers examined are complex.

Traditionally, a web server is classified based on how it handles potentially-blocking network-I/O. In order to achieve high throughput, a server must be able to interleave the processing of thousands of simultaneous connections. For network I/O, it is possible to take advantage of the non-blocking socket operations available on Unix-like operating-systems. With blocking semantics, the calling thread blocks until all the data associated with the call has been buffered or transmitted by the kernel. With non-blocking semantics, only the portion of data that can be immediately buffered in the kernel is transmitted to the client. Then, the calling thread continues and can attempt to send the remaining untransmitted data at a later time. Combining non-blocking operations with an event-polling mechanism like select or poll allows the server to interleave processing of thousands of simultaneous connections. This architecture is referred to as an event-driven server.

Another method for dealing with potentially blocking network-I/O is to use threads. In this approach, if a user-thread blocks waiting on a call, other unblocked threads can execute. In the simple case, each thread only processes a single connection at a time, called thread-per-connection, so thousands of threads may be required. Context-switching among the threads allows for the implicit interleaving of thousands of simultaneous connections. This architecture is referred to as a threaded server.

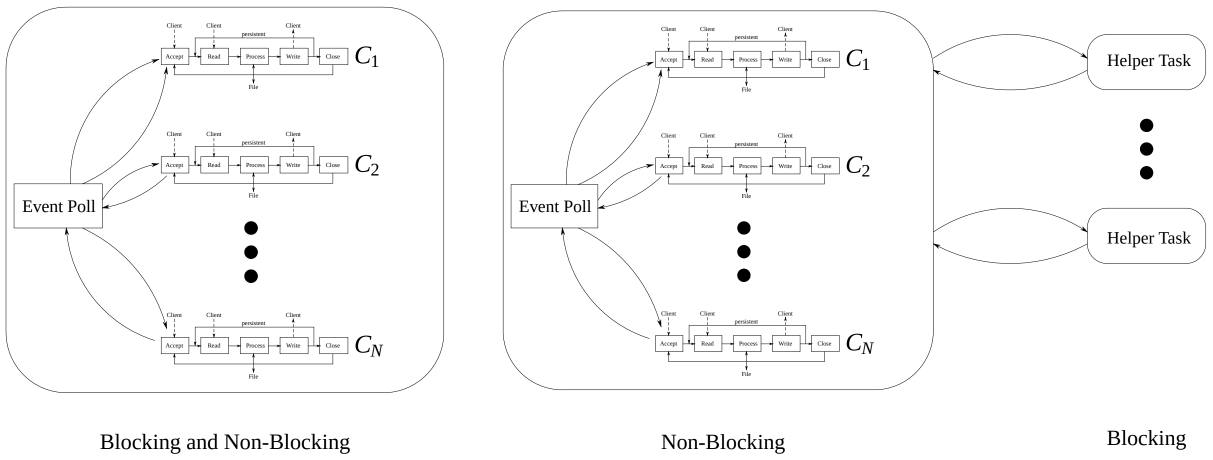
There has been much debate over whether an event-driven or a threaded architecture is better [3, 20, 34, 43, 56]. The argument is event-driven programs tend to be difficult to understand because of complicated control flow and threaded programs tend to have subtle errors and are difficult to debug. A number of libraries have been developed to make event-driven programming easier, e.g., libeel [19] and libasync [20]. Simplifying programming with threads is an area currently receiving a lot of attention but it is a difficult problem and no solution has been found. Unfortunately, this simple classification of server architectures is both inadequate and misleading.

In addition to blocking network-I/O, disk I/O can cause blocking. Asynchronous I/O (AIO) mechanisms exist for some file operations, but performance-critical system-calls like sendfile have no AIO equivalent. Therefore, all server architectures must employ some form of threading to mitigate the effects of blocking disk-I/O. Furthermore, over the past decade, there has been a paradigm shift within hardware architecture from faster processors (more MHz) to parallel processors (multi-threaded, multi-core, multiprocessor) because physical limits in CPU speed and power/cooling are being reached. This change is having a ubiquitous effect on the design of all modern software towards some form of concurrency. As well, servers traditionally labelled as thread-based also involve events. Depending on the type of threading, the event-handling mechanisms are in the thread library, the language run-time library or embedded in the operating-system kernel. Taking a holistic view of the entire architecture, including application, libraries and operating system, all server architectures must incorporate both threads and events to maximize performance. There has been some research into integrating event-driven and threaded programming [3, 20, 27, 32, 62]. While all the servers examined in this thesis use some combination of event-driven and threaded programming, they do not use these approaches.

The following different web-server architectures have been proposed and implemented.

2.2.1 Event-Driven Architecture

Single-Process Event-Driven (*SPED*) architecture [44] is a common approach to implementing a web server, with *μ*server [1] and Zeus [63] as examples of high-performance SPED web-servers. In this approach, a single process (kernel thread) services multiple connections in various stages of processing using non-blocking I/O, see Figure 2.2(a). In the figure, connections are represented by C_i and rounded rectangles represent tasks/threads/processes. Data structures encode the current status of each connection. An



(a) SPED Server (b) AMPED Server

Figure 2.2: SPED and AMPED server

event mechanism such as select or poll is called to determine which connections are currently readable or writable. For each outstanding event, an appropriate event-handler is invoked to process that event without causing the server to block. For example, depending on the type of the event, a new connection may be accepted, a new HTTP request may be read or some data may be written to the socket. Once all the outstanding events are handled, the process is repeated. The complexity of implementing a SPED server comes from the management required to concurrently process and maintain many partially completed connections in differing states with only a single thread (complex finite-state machine).

An interesting extension to this strategy, proposed by Chandra and Mosberger [16] and further investigated by Brecht *et al.* [13], is multiaccept, i.e., to accept multiple connections when the event mechanism indicates that a listening socket is readable. Calling accept multiple times amortizes the overhead of the event-mechanism system-call by establishing multiple connections after each poll. This strategy is employed by the μserver.

One major disadvantage of the SPED architecture is that blocking due to disk I/O significantly decreases performance [44]. Since SPED only involves a single process containing one kernel thread, blocking disk-I/O suspends the entire server until the I/O completes. Based on profiling, μserver spends a large portion of time I/O-blocked on disk-bound workloads. As SPED is single threaded, it cannot take advantage of overlapping CPU execution and disk I/O, nor can it take advantage of multiple CPUs. A natural extension to this approach is to run several copies of a SPED web-server. This approach is called N-copy and was proposed by Zeldovich *et al.* [62]. Not only can this strategy take advantage of multiple CPUs, but it can also be used to deal with blocking I/O. The key is that multiple server-copies are needed per CPU, so when one process blocks due to blocking I/O, another process is usually available to run. As each copy is independent, there is no need for mutual-exclusion/synchronization at the application level; at the kernel level, sharing may occur necessitating locking. However, with N-copy each independent process is listening for connections on a separate socket, requiring additional processing in order to balance requests across the individual servers and to prevent all requests from being sent to only one server.

One approach that extends SPED to deal with blocking disk-I/O is the Asymmetric Multi-Process Event-Driven (*AMPED*) architecture proposed by Pai *et al.* [44] and used in the implementation of the Flash web-server. The idea behind this architecture is to make up for the lack of true non-blocking/asynchronous system-calls for disk I/O on many operating systems by providing helper tasks to handle blocking disk-I/O operations. The basic implementation is to use the SPED approach for all requests that can be serviced from the main-memory file-cache and to pass off blocking disk-I/O operations to helper tasks, as in Figure 2.2(b). Communication between the server task and the helper tasks is done through inter-process communication (IPC), e.g., pipes, so that completion events from the helper tasks are processed by the server task using non-blocking select. Processes may or may not share address-spaces and AMPED relies on the operating system for all process scheduling.

Using helper tasks to perform blocking disk-I/O allows the server task to continue handling requests while disk I/O is in progress. Further benefits include the ability to utilize multiple disk drives more efficiently and more efficient disk-head scheduling. At the core of AMPED is a SPED server, hence it benefits from the advantages of only having a single process managing the processing of connections. Specifically, only a single cache with no mutual-exclusion/synchronization is required as the helper tasks do not modify the cache. Furthermore, the only overhead for long-lived connections is one file descriptor and some application-level data-structures rather than needing an entire thread as with the thread-per-connection approach. Finally, the centralized storage of information simplifies data gathering and only one listen socket is required. The disadvantages of this approach are the IPC overhead and the additional memory required for the helper tasks. The amount of additional memory varies depending on whether address spaces are shared. As well, the AMPED architecture may not scale well on its own with multiple CPUs; while I/O is distributed among several processes, much of the processing is centralized in a single process.

Another approach that builds on N-copy is the Symmetric Multi-Process Event Driven (*SMPED*) architecture by Ren and Wang [37]. SMPED combines a pool of SPED processes with a scheduler process to accept connections and distribute them among the SPED processes and perform load balancing. The server deals with blocking I/O by having a pool of SPED processes whose size is dynamically adjusted based on throughput feedback from each process. This architecture is similar to the Symmetric Multi-Process Event-Driven architecture examined in this thesis (see section 3.8.3), though both were conceived independently.

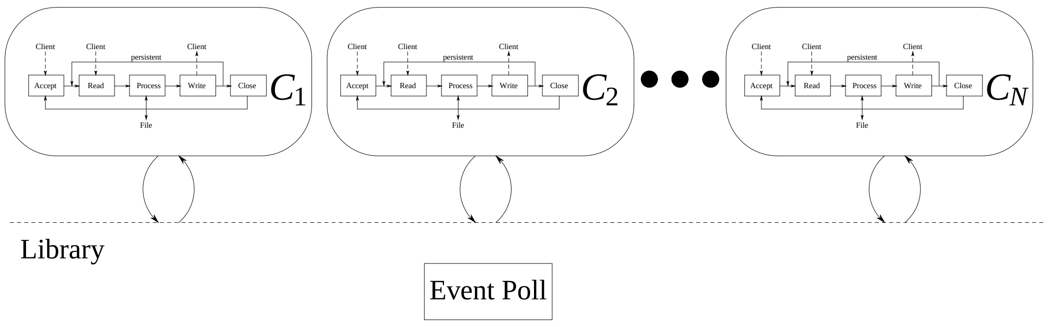


Figure 2.3: Thread-per-connection server

2.2.2 Thread-Per-Connection Architecture

The thread-per-connection architecture is another approach for implementing web servers, e.g., Apache [6] and Knot [57]. In this approach, one thread completely handles a single HTTP request before processing another request, see Figure 2.3. Pai *et al.* [44] distinguish between multi-threaded (MT) and multiprocess (MP) servers. The former involves multiple user-threads in a single address-space and the later involves each thread corresponding to a unique process in a separate address-space. The type of thread-per-connection server examined in this thesis is multi-threaded.

As well, it is possible to create processes or threads dynamically for each new connection or to use a static pool of threads or processes. Dynamically creating threads allows the server to scale depending on the workload. Having a fixed-size pool of threads results in more overhead when the number of connections is small but eliminates a potentially expensive thread/process creation cost for each connection. The thread-per-connection server examined in this thesis uses a static pool of threads.

One advantage of the thread-per-connection architecture is the simplicity gained by making application-level blocking-I/O calls in the server and allowing the interleaving of requests to be handled by context switching rather than having a single thread explicitly manage the state of numerous partially-completed connections. However, there are a number of disadvantages to this approach. First, a web server may need to maintain hundreds or even thousands of simultaneous connections resulting in a corresponding number of threads. Having a large number of threads results in significant memory overhead due to their stacks. Furthermore, there is additional overhead for context switching and from the mutual-exclusion/synchronization required to coordinate access to shared data-structures, like the file cache. Having a separate cache for each thread is impractical when there are a large number of threads as it can lead to significant data duplication. As well, efficiently scheduling a large number of threads is difficult. In addition to the execution costs associated with scheduling, factors such as locking and processor caches must be considered to maximize performance [12, 33, 52].

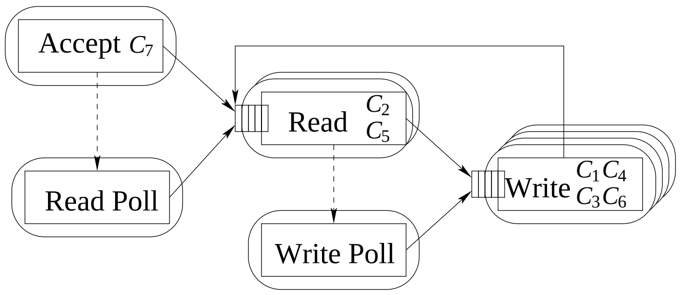


Figure 2.4: Example pipeline server

The underlying thread-library used by a server has a significant affect on performance. The Linux NPTL Pthreads library [22] uses a 1:1 threading model, where each user-level thread is also a kernel thread. With this model, kernel threads can make blocking system-calls without inhibiting the execution of other user-level threads and the server can take advantage of overlapping CPU execution with I/O and multiple CPUs with no additional effort by the application programmer. The problem with 1:1 threading libraries is that they do not scale to support the tens of thousands of threads needed for a high-performance server [57].

Using an M:N user-level threading package is the alternative to allow scaling to thousands of threads. With M:N threading, the threading library typically implements an I/O subsystem built on top of an event mechanism like select or poll, similar to SPED, to handle I/O. As multiple user-threads are multiplexed over a small number of kernel threads, the threading library tries to avoid making blocking system-calls. In order to provide blocking calls for the user threads without actually blocking the underlying kernel threads, system calls are wrapped, and the underlying thread-library deals with blocking by making an equivalent non-blocking system-call. If an equivalent non-blocking system-call is unavailable, the blocking call can be allowed to proceed or handed off to a worker thread, similar to an AMPED helper thread. In either case, a sufficient number of kernel threads are needed to handle blocking I/O while still allowing unblocked user-threads to continue executing. Knot [57] is a thread-per-connection server implemented using the Capriccio thread-package [57], which scales to thousands of threads (see Section 3.8.1, p. 33 for more details).

2.2.3 Pipeline Architecture

The pipeline architecture is another approach for implementing web servers, e.g., [14, 17, 33, 53, 59, 61]. [1] In a pipeline architecture, a server's execution is broken into separate stages, with thread pools to service

[1]Note, while Flux [14] and Aspen [53] are languages and not architectures or servers, the network applications generated by Flux and Aspen have a pipeline architecture.

each stage, see Figure 2.4. The thread pools can be per-stage or shared across multiple stages or the entire application. In the figure, the thread pools are per-stage, with the Read stage serviced by two threads and the write stage serviced by four threads. The number of connections currently being processed by each stage, indicated by C_i, is at most equal to the number of threads, but there may be additional in-progress connections waiting in the queues between stages. A pipeline architecture is often referred to as a hybrid architecture as it explicitly uses threads and events, but the number of threads is fewer than the number of connections. The Staged Event-Driven Architecture (*SEDA*) proposed by Welsh *et al.* [59] is a complex pipeline architecture used to construct the Haboob web-server [59]. SEDA starts as a basic pipeline server by dividing an application into a series of stages. The stages are self-contained but linked by event queues that are used to communicate between the stages. As well, a shared or individual thread-pool is used to service each stage. SEDA then extends the basic pipeline design by adding a dynamic resource-controller that adjusts the thread allocation and scheduling of the stage to meet performance targets. Typically there are only a small number of threads per stage, and an application consists of a network of stages. For example, the Haboob server contains the following stages: Socket Accept, Socket Read, HTTP Parse, PageCache, CacheMiss, File I/O, HttpSend and SocketWrite. An HTTP request is passed from stage to stage as it is processed through the pipeline.

There are several advantages to using a pipeline server with self-contained stages. First, the modularity of this approach allows stages to be reused in several applications and allows for independent load-management. As well, it makes debugging easier and facilitates performance analysis. For example, the size of the queues connecting the stages show how well the application is running and help to identify bottlenecks. Third, using finite event-queues makes it easier to perform load shedding as requests can be terminated at any stage.

While pipeline servers like SEDA have the complexity of both events and threads at the application level, they also benefits from some advantages of both approaches and allow better control over each. SEDA utilizes the efficiency of the event-based approach to reduce the number of threads significantly below the number of connections. Furthermore, with multiple threads spread across the various stages, pipeline servers can also benefit from the overlapping of I/O and CPU execution as well as multiple CPUs. Pipeline servers also allow for cohort [33] or convoy [60] scheduling, where tasks with similar operations are scheduled together to allow for better data and instruction locality. As each stage in a pipeline is a grouping of threads executing similar operations, scheduling these threads to execute together achieves this objective.

The obvious disadvantage of this approach is that each connection incurs the overhead of passing through a number of stages in the pipeline. This process involves an enqueue/dequeue for each stage and likely a context switch as a different thread processes the request at each stage. However, this overhead can be reduced by keeping the pipeline small and by having each thread handle several requests before it is preempted, i.e., amortizing a context switch across several requests keeps the overhead manageable. Finally, mutual-exclusion/synchronization is needed to protect the queues as multiple threads on different processors may try to access the queues simultaneously, increasing complexity and runtime cost.

2.3 Uniprocessor Performance Comparisons

One of the objectives of this thesis is to compare current state-of-the-art web-server architectures on a uniprocessor. In this section, a number of previous comparisons for web servers on a uniprocessor with a static workload are discussed. The first comparison discussed is similar to the comparison of uniprocessor architectures presented in this thesis; several servers are examined across various workloads. However, a number of changes have occurred since that comparison, making a new comparison necessary, including new server architectures, new implementations for existing server architectures, new operating-system facilities and faster networks. As web server architectures have been introduced or refined, performance among new and existing servers has been compared. The subsequent discussion presents the evolving comparison picture as these changes occurred.

Pai *et al.* [44] implemented Flash, based on the AMPED architecture, and performed tests on various server architectures for different workloads on a uniprocessor connected with multiple 100 Megabit per second (Mbps) Ethernet interfaces. Several architectures were implemented from a common code-base, including AMPED, SPED, MP and MT. They show that architecture is not a significant factor for trivial tests with a single file. For more realistic workloads, SPED, AMPED and MT have approximately the same throughput when the file set fits into cache and the MP server has approximately 15–30% lower throughput. As the workload becomes more disk bound, the performance of all the servers declines but only the throughput of the SPED server drops significantly as it lacks additional threads to deal with blocking disk-I/O. A final test with hundreds of concurrent connections, to model more realistic WAN conditions, shows the performance of the MT server gradually declines and the performance of the MP server declines more sharply as the number of concurrent connections increases, while the performance of SPED and AMPED remain flat. They demonstrate the importance of threads for disk-bound workloads and show the potential scalability problems with thread-per-connection servers.

Welsh *et al.* [59] also performed tests on various server architectures to compare the performance of SEDA. They implemented a SEDA server called Haboob in Java and compared it to Flash, which is AMPED, and Apache, which is thread-per-connection. Flash and Apache are both implemented in C. The workload tested is heavily disk bound as the size of the file set is 3.31 GB but the page cache is only 200 MB. In the experiments, throughput is measured as the number of client connections are varied. Both the AMPED server and the thread-per-connection server peak with fewer connections and lower throughput than the SEDA server; Haboob has approximately 15–20% higher peak throughput than the other two servers. None of the servers experience degradation in throughput as the number of connections increases; however, Apache caps its connections at 150 and Flash at 506, while Haboob supports the maximum of 1024 connections tested. As well, Flash and Apache have a much larger distribution of response times, with large maximum values. Although the machines were connected with gigabit Ethernet interfaces, the throughput is low because the workload is heavily disk bound and the number of connections is small.

von Behren *et al.* [57] compare the performance of Knot, Apache and Haboob. The workload for their experiment is heavily disk bound since the cache size is limited to 200 MB and the file set is 3.2 GB. Their experiment measures throughput as the number of connections is increased to the tens of thousands. Up to 100 connections, the performance of the servers is approximately the same. Beyond 100 connections, the performance of Haboob and Knot is about the same, and approximately 30% higher than the peak performance of Apache. As the number of connections approaches 10,000 and higher, the performance of Haboob and Knot drop until they reach the same level as Apache.

Brecht *et al.* [13] compare the performance of Knot and *μ*server with an in-kernel web-server, TUX [36, 48]. They used two static workloads, an in-memory SPECweb99-like workload and a one-packet workload on a uniprocessor with two gigabit Ethernet interfaces. Though their results are expressed in replies per second, larger replies per second typically indicate higher throughput, especially for the one-packet workload. Based on the experiments, *μ*server has 40–50% higher throughput for the in-memory SPECweb99-like workload and 50–60% higher throughput for the one-packet workload compared to Knot. In this comparison, *μ*server uses select while Knot uses poll and *μ*server has a maximum connection limit of 15,000 while Knot is configured to use only 1000 threads (connections). As expected [29, 49], the in-kernel server has the best performance, but *μ*server has performance close to TUX for the SPECweb99-like workload.

Burns *et al.* [14] compare Haboob and Knot with servers of different architectures built using the Flux language. The workload for their experiments is in-memory, consisting of a 32 MB static file-set on a uniprocessor with a gigabit Ethernet interface. Their focus was on stressing CPU performance. In their experiments, Knot has approximately two times the throughput of Haboob. The event-based Flux server and the thread-pool based Flux server both had performance comparable to Knot but the Flux server with dynamic threads had the worst performance.

Park and Pai [46] compare the performance of Flash, Apache and Haboob to servers modified to use connection conditioning. With connection conditioning, requests pass through filters, to provide security and connection management, before being passed on to the server. Their experiments were on an inexpensive uniprocessor with a gigabit Ethernet interface and involved static workloads. For in-memory workloads, their experiments show that Flash has the best performance, with Apache having lower performance and Haboob having poor performance. As the workload shifts to more disk bound, the performance difference among the servers reduces until all the servers have approximately the same performance except Haboob whose performance is relatively flat and lower than the other servers for all workloads.

As shown by the previous discussion, a number of changes occurred since the original performance comparison by Pai *et al.* [44]. Unfortunately, the subsequent comparisons have not been as thorough or consistent. In most cases, only one type of workload is examined, either in-memory or disk bound. Since relative server performance can change based on workload, it is hard to get a complete picture of server performance from a single workload. As well, implementation and configuration differences among the servers make architecture comparisons difficult. For example, the number of connections supported by the various servers in the comparison by Welsh *et al.* [57] differs and in the comparison by Brecht *et al.* [13]. Another example is that the implementation languages are not consistent; Haboob is implemented in Java while many of the other servers are implemented in C or C++. These differences can have a large effect on performance. Finally, some of the results are contradictory. For example, in some experiments Haboob performs well compared to Apache or Flash and in other experiments extremely poorly. As a result, it is difficult to determine whether relative performance differences are due to architecture or other factors, such as, language, implementation, tuning, memory footprint, workload, etc. Given these short-comings, it is reasonable to perform a thorough comparison of current state-of-the-art web servers on a uniprocessor across various workloads. The comparison in this thesis attempts to be fair, by addressing the problems discussed in this section, and uses current best practices like sendfile and epoll.

Note, the publication [45] is a preliminary version of the moderate disk-I/O uniprocessor-workload presented in Chapter 3. However, there are a number of differences that make the results in that publication incomparable to those in the thesis. The differences in this thesis include 1.4 GB of system-memory instead of 2 GB for the moderate workload, the use of epoll, a Linux kernel patched to fix a caching problem, a new sendfile implementation for Knot and large changes to WatPipe based on adding epoll support.

2.4 Multiprocessor Performance Comparisons

Another objective of this thesis is to compare the performance of multiprocessor server-architectures. Little research has focused on web-server architectures specifically targeted for multiprocessors. Three previous comparisons for web servers on multiprocessors with a static workload are presented.

Zeldovich *et al.* [62] compare the performance of a number of web servers. In addition to AMPED (Flash) and thread-per-connection (Apache), they also measured the performance of N independent copies of a single-process web-server, where N is the number of CPUs. For a balanced workload, they suggest that the N-copy server represents an upper bound on performance as each server process runs independently. Note that Flash also creates one independent server copy per CPU on a multiprocessor, similar to N-copy, except the servers share a listen socket. The purpose of their experiment is to measure the multiprocessor speedup for various architectures. The workload is in-memory with a warmed file-cache. The experiments show the performance of Flash is better than Apache for one processor, but as the number of processors increases to four, the difference in performance shrinks. However, the performance of both servers is less than the N-copy server. An interesting result of the experiment is that the speedup of all the servers is small, especially beyond two CPUs. Similar to the multiprocessor comparison in the thesis, the various servers are compared to an N-copy server, with the N-copy server having the highest throughput for the in-memory workload. However, Zeldovich *et al.* do not include a pipeline server in their comparison, affinities and partitioning (see Section 4.4, p. 117) are not considered and only one workload is examined.

Choi *et al.* [17] propose a pipeline architecture for multiprocessor systems and compare the performance to AMPED, SPED, MP and MT architectures using a simulator. In their architecture, processing an HTTP request is broken into pipeline stages, with each stage serviced by a single thread. A single pipeline is referred to as a pipelined thread-pool, and the server consists of multiple pipelined thread-pools in a single address-space. For the pipeline-server experiments there is one pipelined thread-pool per CPU, for the AMPED and SPED server experiments there is one server copy per CPU and for the MT experiments there are 32 or 64 threads per CPU. Based on the simulations, they conclude the memory footprint of a server is important in determining the performance of a server especially as the number of CPUs becomes large. Hence, servers with separate address-spaces scale poorly compared to servers with shared address-spaces on multiprocessors when using an N-copy approach. As well, they found that contention became a problem for MT servers as the number of threads becomes large but not for the other servers because the number of threads sharing data is small.

There are a number of similarities between the work done by Choi *et al.* and Chapter 4 of this thesis. Both examine pipeline and N-copy servers under different workloads, and find memory footprint to be an important factor in server performance. However, there are also a number of differences. The most important difference is their experiments are run on a simulator, which allowed testing more workloads, while the experiments in this thesis are actually run on a multiprocessor. The experiments run by Choi *et al.* are a reasonable starting point, but do not take into account affinities and partitioning, scalability issues (outside of file-cache locking), cache coherency or many other factors that come into play on a real system. However, the advantage of running on a simulator is the number of CPUs (processing elements) are scaled from 1 to 15, a larger range than examined in the thesis. As well, the pipeline server proposed by Choi *et al.* is different from the architecture in this thesis. Differences include the number and function of stages in the pipeline, the use of sendfile in the thesis, the number of threads per stage, etc. As well, Choi *et al.* do not discuss the mechanism used to poll for events in their simulation. Based on experiments in Chapter 4, their strategy of only using one thread per CPU to read data from disk is insufficient in the presence of disk I/O. As well, their experiments show surprisingly little difference in throughput between the SPED and AMPED servers when there is disk I/O.

Upadhyaya *et al.* [53] compare the performance of Flash, Haboob and a server developed using Flux [14] with a pipeline server they developed using Aspen. The workload for the experiments consists of a 3.3 GB static SPECweb99-like file-set on a 4 CPU multiprocessor with four gigabit Ethernet interfaces. Since the server has 4 GB of memory, the workload is probably in-memory or has a small amount of disk I/O. Their experiments show that the pipeline Aspen server is scalable as the number of concurrent client-connections increases, eventually peaking with the highest throughput. Flash and the Flux server achieve higher throughput for a smaller number of concurrent client-connections but their throughput peaks quickly, levels off and is eventually surpassed by the Aspen server. The performance of Haboob is much lower than the other servers. Flash and Haboob use an application file-cache and Aspen uses sendfile. Flux does not use an application file-cache, but it is unclear if it uses sendfile. Similar to the experiments in the thesis, the pipeline server performs well. However, the use of sendfile by Aspen gives it a performance advantage. Unfortunately, there is not enough information to determine what factors are affecting the performance of the various servers. As well, only one workload is tested and affinities and partitioning are not considered.

Voras and Zagar [58] compare the performance of SPED, SEDA, AMPED and SYMPED on the mdcached application, a memory database for web caches. While mdcached is not a web-server, it is a high-performance network server. The workload for the experiments is in-memory on a 8 CPU machine. However, the clients and server run on the same machine so only 4 CPUs are dedicated to the server. The experiments show that SYMPED has the best performance, followed by SEDA and AMPED with similar performance and SPED has the worst performance. In their experiments, SYMPED had a linear increase in performance up to 4 threads. These experiments cannot be directly compared with the experiments in the thesis because the operating system, test application and workload are different from those used in the thesis; however, the result that SYMPED performs well for in-memory workloads is consistent with the experiments in the thesis. The authors do not provide an explanation for SEDA's performance.

Web-server experiments on multiprocessors have been relatively limited and have tended to ignore important issues like affinities and partitioning. The non-simulator experiments have focused exclusively on in-memory workloads, but workloads requiring disk-I/O must be examined as they are more realistic. As well, the server throughput for these experiments is relatively low, so it does not stress the server architectures. Similar to the uniprocessor experiments, it is difficult to determine the factors causing the performance differences among the various servers. The simulator experiments are a reasonable starting point, but must be verified on an actual machine.

2.5 File-System Cache

The Linux file-system cache is important for the experiments in this thesis. For example, file data is transmitted using sendfile via the file-system cache. As well, its size dictates the amount of disk I/O required during an experiment; as the level of disk I/O increases, the throughput of a server decreases. The file-system cache is global to the operating system and shared by all processes and threads. This section presents a brief description of the Linux file-system cache.

The Linux file-system cache is used to cache data from disk, including meta-data, directory information and file data. Data read from disk is typically stored and accessed from the file-system cache because disk access is slow compared to memory access. The cache is made up of pages, where a page is a fixed-size block of memory. Cached data is stored indefinitely as long as there is sufficient free memory. The size of the file-system cache is limited by the amount of free memory. If a section of a file is accessed that is not currently in the file-system cache, it must be read into the cache. However, if there is no free memory, some existing file data must be evicted so the new file data can be read. A Least Recently Used (LRU) algorithm is used to determine which pages of file data should be evicted from the file-system cache. While changes to disk data also proceed through the file-system cache, this operation is not used in this thesis, so it is not discussed further as it is irrelevant to the experiments.

Since disk access is slow and most file access is sequential, the operating system typically reads a large amount of adjacent data for a single disk read into the file-system cache. In addition, if the operating system detects that file access is proceeding sequentially, it also performs additional read-ahead by reading the next chunk of sequential file data from disk before it is requested by the application. Performing larger reads improves disk efficiency and using read-ahead reduces the amount of time an application waits for disk I/O as the data has already been requested. If the operating system detects that file access is not sequential, then read-ahead is disabled.

2.6 API Improvements

Many of the servers discussed in this chapter do not take advantage of newer operating-system calls. This section reviews some recent application-programming-interface (API) extensions applicable to high-performance servers.

2.6.1 Scalable Event-Polling

Event-polling mechanisms are important for high-performance servers. Event-driven architectures such as SPED and AMPED use event polling to determine what actions to perform next. Many thread libraries use event polling to minimize blocking, allowing multiple user-threads to be multiplexed over a small number of kernel threads. Banga and Mogul showed that traditional interfaces like select and poll do not scale well for a large number of file or socket descriptors [8]. In the context of a web server, each connection corresponds to a socket descriptor.

One problem with select and poll is that the cost of the operation is proportional to the number of descriptors, not the number of available events. These costs include O(n) copying costs for arguments such as bitmasks or arrays into the kernel and back to user space and O(n) costs to scan the list of descriptors to find outstanding events. Another problem is the calls are stateless, so each call requires the interest set as an argument. Hence, multiple calls on the same set of file descriptors require the interest set to be copied into the kernel each time, versus the kernel retaining the interest set.

Research on more appropriate event-polling mechanisms for high-performance servers handling a large number of simultaneous connections has resulted in a number of new event-polling mechanisms [9, 16, 47]. These new event-polling mechanisms have been incorporated into various operating systems, including /dev/poll on Solaris, Kqueue [35] on FreeBSD and epoll on Linux. I/O completion ports offer similar functionality under Windows [9, 55].

The idea behind these mechanisms is similar to the scalable event-mechanism proposed by Banga *et al.* [9]. In their mechanism, rather than having event polling be stateless, the kernel stores the interest set on behalf of the application. The application program calls an event-polling routine to retrieve events associated with the stored interest-set or calls update routines to modify the interest-set stored in the kernel. Their mechanism is both efficient and scalable; it reduces the amount of data copying required for polling and the performance of the event-poll calls depend on the number of available events and not the size of the interest set. One disadvantage to this approach, however, is that a copy of the interest set must be kept inside the kernel, potentially resulting in additional memory overhead as a copy of the interest set likely exists in the application too. As well, each change to the interest set requires an expensive call into the kernel, offsetting some of the advantages, especially if the interest set changes frequently [26].

All the servers examined in this thesis take advantage of epoll, with functionality similar to the event mechanism proposed by Banga *et al.* [9]. There are two modes of operation for epoll: level-triggered and event-triggered. With level-triggered semantics, as long as there are events pending for a descriptor, polling that descriptor returns, indicating events are available. With edge-triggered semantics, polling a descriptor only indicates available events when the event first occurs; subsequent polls of the same descriptor do not indicate that events are pending until all events for that descriptor are drained. Typically, the application continues reading or writing to a descriptor until EAGAIN is returned before it can expect to be notified of further events for that descriptor.

Gammo *et al.* [26] compared the performance of epoll, select and poll. They show that edge-triggered epoll performs better than level-triggered epoll for web servers under various loads and that edge-triggered epoll performs equivalently or better than select or poll for a variety of workloads. As well, similar to Chandra and Mosberger [16], they show that select and poll perform reasonably under high load for certain workloads provided multiaccept is used. The servers examined in this thesis use edge-triggered semantics for epoll.

2.6.2 Zero-Copy Transfer

Originally, web servers maintained a cache of file data in the application. File data is read from disk into the application file-cache and then the write system-call is used to transmit the data to the client. Hence, the file data is read from disk into the kernel file-system cache and subsequently copied into the application file-cache. Then, the data must be copied from application file-cache back to the kernel every time a file is requested and transmitted to a client. This method is inefficient as it incurs at least double copying every time file data is transmitted to a client, and possibly more copying if data is evicted from the application file-cache due to size restrictions.

To reduce the inefficiency, operating systems offer zero-copy transfer methods: sendfile on Solaris, Linux and FreeBSD and TransmitFile on Windows. With zero-copy transfer, the application does not need to maintain an application file-cache. Instead, the file data in the kernel file-system cache is utilized to transmit the file to the client, eliminating the overhead of copying the file data between user space and the kernel. Both, Joubert *et al.* [29] and Nahum *et al.* [40] show the performance benefits of zero-copy transfer. One potential downside of using zero-copy transfer is that the kernel decides which files remain in the cache, not the application.

2.6.3 Asynchronous I/O

While sockets can be placed into non-blocking mode, that option does not exist for disk I/O. Instead, operating systems are starting to offer support for asynchronous disk-I/O. Like non-blocking socket-I/O, with asynchronous I/O, an application makes a request for disk I/O but does not block waiting for the I/O to complete. At some point in the future, the kernel notifies the application once the I/O completes. Currently, there is no unified event-mechanism supported by Linux; ideally, the kernel notifications should be tied into an event mechanism like epoll, so all I/O can be managed by the application through a single interface. The advantage of this approach is that only a single thread is needed on a uniprocessor as an application never needs to block waiting for socket or disk I/O. However, multiple threads or processes are still needed to achieve parallel execution on a multiprocessor. Hu *et al.* [28] show the advantage of using asynchronous TransmitFile in Windows NT under heavy load for large file transfers compared to multi-threaded servers using synchronous I/O.

Unfortunately, the Linux kernel does not offer full support for asynchronous I/O [21]. For example, as mentioned previously, asynchronous sendfile does not exist. In the absence of kernel support, it is possible for an application to use kernel threads (helper tasks) to simulate asynchronous I/O, i.e., a set of kernel threads are used to submit disk-I/O operations on behalf of the application and notify the application once the I/O completes. However, this mechanism is less efficient than a kernel implementation as there are overheads related to mutual-exclusion/synchronization as well as memory and scheduling overheads resulting from additional threads.

Another approach is for the operating system to provide a general mechanism, like Scheduler Activations [5] or First-Class User-Level Threads [38], that allow thread libraries or applications to implement asynchronous I/O operations on top of existing synchronous I/O operations. Elmeleegy *et al.* [23] propose Lazy Asynchronous I/O, a user-level I/O library to allow event-driven servers to deal with blocking I/O operations. The library provides a polling mechanism to allow both socket and file I/O to be handled in a consistent manner. The advantage of this approach over traditional asynchronous I/O is that I/O continuations are only created if an operation actually blocks. Chanda *et al.* [15] developed ServLib, a thread library that provides an M:N threading library built using asynchronous I/O. While most M:N threading libraries contain an I/O subsystem to transparently deal with blocking calls so the entire application does not blocking waiting for I/O, ServLib provides this functionality by building its I/O subsystem using asynchronous I/O. They show that M:N threading with asynchronous I/O offers better performance than other threading models for I/O intensive multi-threaded servers like web servers. Both Lazy Asynchronous I/O and ServLib are built on top of Scheduler Activations. The drawback of Scheduler Activations is that a new kernel thread is spawned every time an operation blocks, which could result in high overhead. As well, many operating systems, including Linux, do not provide support for mechanisms like Scheduler Activations.

2.7 Summary

Web servers are complex software applications. They must handle network and disk I/O efficiently and scale to thousands of concurrent connections. Previous research has focused on improving web servers serving static content. These improvements have included new architectures, implementations and operating system facilities. Web server architectures include AMPED, thread-per-connection and pipeline. High performance implementations of these architectures are Flash for AMPED, Knot for thread-per-connection and Haboob for pipeline. Newer operating system facilities include zero-copy transfer and scalable event-polling, e.g., sendfile and edge-triggered epoll, respectively.

The current performance picture for web servers is unknown because a thorough comparison encompassing all these facets is lacking. Furthermore, much of this work has focused on the uniprocessor domain. Chapter 3 presents a comparison of current state-of-the-art web servers on a uniprocessor across various workloads. Once the uniprocessor situation is understood, Chapter 4 continues the comparison into the multiprocessor domain.

Chapter 3

Uniprocessor Web-Server Architectures

This chapter examines various server architectures run on a single processor with different workloads. The goal of this chapter is to compare the performance of the server architectures and to see how their performance changes under different workloads. A spectrum of workloads, ranging from in memory to disk bound, are explored. The varying workloads are achieved by reconfiguring the server with different amounts of memory. All other factors are kept the same, including the file set, the log files used by the clients, the network configuration, etc.

For each workload, similar parameters for each server are tuned for best performance. All experiments are verified (see Section 3.3) to ensure the server is providing fair service to the clients and only those experiments passing verification are included. The best configuration for each server is profiled to examine any differences and similarities among the architectures and for different implementations within an architecture.

3.1 File Set

The file set used for the experiments in this chapter is static and generated using the SPECweb99 [50] file-set generator. The choice of a SPECweb99 file-set allows the results to be comparable to the large body of previous work [13, 46, 53, 57, 59, 62], which uses a static SPECweb99 file-set. In many cases, these file-sets are also of a similar size as the file-set in the thesis. More recent versions of the SPECweb benchmark exist, and it would be interesting to test the server architectures on these and other workloads, but that is beyond the scope of this thesis. The various workloads for this chapter are generated by keeping the file set the same and reconfiguring the server with different amounts of memory. This strategy has the advantage of keeping one additional variable consistent among the experiments.

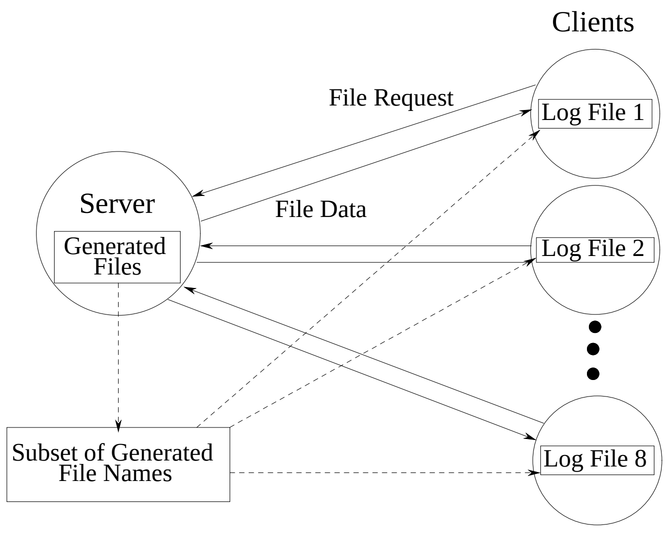


Figure 3.1: From file set to HTTP requests

The SPECweb99 generator produces a number of directories each containing files with the same profile. The files are classified into four classes each consisting of nine files, with all thirty-six files having unique sizes. The sum of the file sizes in a single directory equals 5,123,580 bytes. Class 0 files range in size from 102 bytes to 921 bytes. Class 1 files range in size from 1024 bytes to 9216 bytes. Class 2 files range in size from 10,240 bytes to 92,160 bytes. Class 3 files range in size from 102,400 bytes to 921,600 bytes. The size of the files is constant across the directories. Directories are generated to make up a file set of the desired size. While each directory contains 36 files (4 classes $\times$ 9 files), the contents of each file differs.

Figure 3.1 shows the relationship among the file set, server and clients. For the experiments in this chapter, 650 directories were generated, resulting in a file set of approximately 3.1 gigabytes in size on the server. However, the size of the generated file-set overestimates the size of the actual file-set used in the experiments. The size of the actual file-set is based on the subset of files a client might actually request.

Each client consists of a copy of the httperf load-generator [39] running with a log file of requests. The load generator simulates a number of users, based on the desired request rate, by establishing multiple concurrent connections to the server. The main advantage of httperf is its ability to generate overload request-rates, dealing with the inability of the SPECweb99 load-generator to generate overload conditions as discussed by Banga and Druschel [7]. Individual clients all have distinct log files, with the log files generated based on the specifications of the file set and the requests in a log file are designed to conform to a Zipf distribution [64]. Each log file contains a number of persistent HTTP connection sessions, where each session is a request for one or more of the files. The log files include both active and inactive off

% Reqs	Memory Size (MB)	Max File size (B)
10	0.5	409
20	0.8	512
30	1.6	716
40	4.8	3072
50	8.4	4096
60	9.9	5120
70	12.2	5120
80	20.1	7168
90	94.6	40,960
95	127.2	61,440
100	2196.0	921,600

Table 3.1: Cumulative amount of memory required for requests when sorted by file size

periods to model browser processing times and user think times [10]. For this chapter, the log files for each client have an average session length of 7.29 requests and all the requests are for static files. Due to the way the log files are generated, not all the files in the file set are requested. The log files request 2.1 GB, consisting of 21,396 files across all 650 directories of the file set. Hence, the size of the requested file set is approximately 2.1 GB.

Interestingly, even this value does not represent the effective file-set for a specific experiment. The effective file-set during an experiment is based on the requests the server actually processes. During the experiment, some client requests may timeout causing not only the request itself to not be processed by the server but also all the subsequent requests in that session. Hence, it is possible that not all files in a client log-file may be processed by the server, meaning that the effective file-set experienced by the server may vary from run to run. Nevertheless, in the remainder of the thesis the term file set is used to refer to the file set based on all possible client requests as it gives a reasonable approximation of the file set experienced in a given experiment and is the file set that a server would handle if it was able to process all client requests (i.e., no timeouts).

The various workloads in the chapter are generated by keeping the file-set the same and reconfiguring the system with different amounts of system memory. Given that the server machine used for these experiments has 4 GB of physical memory, a file-set size of 2.1 GB fits entirely in memory and still leaves enough memory for the operating system and application. For the experiments requiring moderate disk activity, the system is booted with 1.4 GB of memory and for high disk activity, the system is booted with .75 GB of memory.

When a server receives a valid file-request, it sends back the file data to the client. The question of how disk bound the file set is depends on the amount of memory available on the server and the pattern of requests. Table 3.1 shows the amount of memory required to satisfy requests as the files size increases.[1] For example, 60 percent of requests are for files of size 5120 bytes or less and the sum of the sizes of the unique files contained in those requests is 9.9 megabytes. Due to the Zipf distribution, only a small amount of memory is needed to service a significant percentage of requests; 50 percent of the requests comprise 8.4 MB of the file set and 95 percent of the requests comprise only 127.2 MB of the 2.1 GB file-set.

3.2 Response Time

In order to simulate more realistic workloads, all client requests must be serviced within a certain time. If a request is not completed within that time frame, the client times out and closes the connection. This behaviour models a real user who only waits for a relatively small amount of time for a web page to load. This response time value may also vary from user to user, and a user's response-time tolerance may vary based on the web site being accessed.

For the experiments in this chapter, a single timeout value of 10 seconds is chosen and applied to all requests. 10 seconds is reasonable based on the following observations. First, in Windows XP, the TCP/IP stack is tuned to wait 9 seconds when trying to establish a connection before timing out. Second, Nielsen [41] recommends that 10 seconds is the upper limit on acceptable response time based on a variety of user studies examining human-computer conversational interactions conducted from 1960 to 1980. While 10 seconds sounds reasonable, based on expectations of current broadband users and the fact that the networks in these experiments are running at gigabit speeds, a timeout value of 10 seconds is likely rather generous. In fact, newer studies [30] suggest this value may be lower for certain types of sites. However, even with a broadband connection, downloading a 1 MB file (approximate size of the largest file in the file set) can take a few seconds. Hence, 10 seconds is a reasonable compromise.

Timeouts are enforced using the timeout parameter to httperf. However, this parameter is used for two purposes. While establishing a connection, the timeout parameter is the amount of time the server has to respond to a SYN. This value is in line with the Windows XP connection timeout mentioned above. Once a connection is established, it becomes the timeout value for the entire persistent HTTP connection session. All the files in the session must be completely sent within this amount of time.

[1]Files of size 5120 B span across 60%–70% of requests

3.3 Verification

The servers compared in this thesis are verified based on two criteria. First, it is necessary to ensure correctness, i.e., that the server is sending valid data to the clients. This step is accomplished by having each client compare the bytes returned by the server with a copy of the actual file requested. A server is running correctly if all the data matches. Due to the significant overhead required to compare the data sent by the server, only a few representative request rates are run to verify the correctness of the server; correctness verification is disabled during the performance experiments. When there is a significant change to the server, this verification is repeated to make sure the server is still sending valid data.

Second, it is necessary to ensure adequate performance, i.e., to verify the server is achieving an acceptable response with respect to the entire range of client requests. This requirement creates a level playing field and allows for a fair comparison of the servers. The criteria used to establish this range is that 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, e.g., requests for large files.

This verification step considers both the individual clients and the aggregate of the clients. Three criteria are used to determine if a server is operating reasonably. These criteria focus on the percentage of requests that timeout both cumulatively across all files and for each file size. Client requests that timeout before being accepted or read by the server are not counted because factors external to the server are controlling this decision; in this case, the server has not seen the requests, and hence, is not explicitly rejecting specific requests. While client timeouts are permitted across all file sizes, verification ensures that each size receives a reasonable level of service. The criteria are as follows:

- The maximum percentage of timeouts for all files does not exceed 10%.
- The timeout percentage for each file size is below a certain threshold. For individual clients, the threshold is 5% and the aggregate for all clients is 2%.
- For each file size, the timeout percentage is not larger than the mean timeout percentage of all files plus a threshold. Again, for individual clients the threshold is 5% and the aggregate for all clients is 2%.

The first check ensures that there is not an excessive number of timeouts in general. The second check ensures that no individual client experiences a disproportionate number of timeouts. The third check ensures that no file size experiences a disproportionate number of timeouts. This check is similar to that performed in SPECweb99 to verify that approximately the same quality of service is afforded to files of different sizes. Based on experience, these checks may also help to determine if a server is operating incorrectly or if there is a problem with the experimental environment.

Without this check, various techniques can be employed to improve performance. For example, it was found that higher throughput could be obtained for certain workloads by allowing requests for the largest files to timeout. Due to the fact these workloads use the Zipf distribution, smaller files are accessed more frequently and hence, tend to remain in the file-system cache. Thus, giving priority to smaller files can result in higher throughput and smaller response times because there is less disk I/O.

The verification step to ensure adequate performance does not effect server throughput as it is run after the experiment finishes. All experiments undergo this verification unless otherwise indicated and results are only included for experiments that pass verification for all request rates. In instances where experiments did not pass verification, it is most often due to servers not being able to respond to requests for large files prior to the client timing out.

3.4 Tuning

When implementing a server, a number of design choices are available, with the biggest being the server architecture. Once a server architecture is chosen, a number of smaller design choices still exist. Many of these choices are made either through trial and error or by following best practices. These choices can include how the data is transmitted (sendfile vs write), the event mechanism (select vs poll vs epoll), etc. For each architecture examined, best practices are used to implement a server.

However, even when best practices are used, there are a number of configuration parameters that affect a server's performance with respect to different workloads and hardware, e.g., maximum number of simultaneous connections, number of threads, etc. Ideally, these parameters should be automatically adjusted by the server according to the dynamic workload. However, *auto-tuning* is a challenging problem [11]. In this thesis, tuning is performed systematically by hand.

The basic idea is to run experiments and measure the performance of the server as the parameters of interest are varied. By observing how the performance of the server changes with different tunings, it is possible to find a combination of parameters that result in the best performance. Tuning is done independently for each server and workload combination. As the number of possible tuning parameters for each server is quite large, a subset of parameters is selected. Based on my experience in running a large number of different server experiments, a subset of appropriate parameters was chosen that have the most significant affect on performance.

The parameters chosen in general are the maximum number of simultaneous connections supported by the server, level of concurrency and blocking versus non-blocking sendfile. The terms blocking sendfile and non-blocking sendfile refer to whether a socket is in blocking or non-blocking mode when sendfile is called. For Knot using an application level cache (knot-c, see Section 3.8.1), the size of the application file-cache is the third parameter tuned; knot-c does not use sendfile. After selecting the type of sendfile or cache size for knot-c, a range of values for both the maximum connections and the level of concurrency are 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. The original ranges are chosen to be sufficiently large so that the entire spectrum of performance is covered.

The initial set of experiments for each server are analyzed to get an indication of the general vicinity resulting in good performance. Once this area is found, a more fine-grained matrix can be used to find the combination of parameters that resulted in the best performance. This procedure can recurse as many times as necessary but typically only two or three levels are necessary.

Each experiment consists of running the server for several rates, with each rate taking about 5 to 8 minutes to complete. Client requests are generated using httperf as the load generator. The total request rate is evenly divided among each copy of httperf, which is responsible for generating its portion of the load using its associated log file. A client traverses the log file one or more times based on the length of time the experiment lasts and the desired request rate. For the experiments in this chapter, the request rates range from 6000 requests per second to 30,000 requests per second. This range is wide enough to find a server's peak throughput and to measure its throughput after saturation. In the initial tuning, an experiment consists of five request rates and for the fine-grained tuning the number of request rates is increased to eight. The selected rates are run in increasing order and the server is restarted between each rate. There are 2 minutes of idle time between rates and between experiments to allow any connections in the TIME-WAIT state to be cleared. As a result, 45 to 75 minutes are required to produce a single line on a graph plotting multiple rates.

3.5 Environment

The 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. Each client machine runs two copies of the workload generator. They run a 2.6.11-1 SMP Linux kernel, which permits each client load-generator to run on a separate CPU. The server machine is identical to the client machines except that it contains 4 GB of RAM and a single 3.06 GHz Xeon CPU. For all experiments, the server runs a 2.6.16-18 Linux kernel in uniprocessor mode, i.e., the kernel is built with SMP disabled.

The 2.6.16-18 Linux kernel on the server has been modified to fix a problem with caching in the kernel that I found and fixed. The file-system cache in the 2.6.16-18 kernel is designed to prevent a single, sequential non-page-aligned read of a large file from invalidating a large portion of the file-system cache.

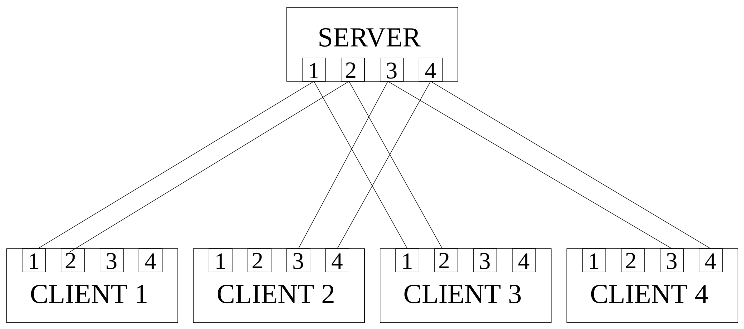


Figure 3.2: Networking between server and client machines

The problem occurs because the mechanism to detect this behaviour is too coarse; multiple consecutive accesses to the same page in the file-system cache do not update the access flags for that page. Only when a different page in the file is accessed are the access flags updated. This logic causes files that are less than or equal to the size of a single page to never be marked as accessed after their first access. Hence, these pages are always ejected from the cache regardless of how often the file is accessed. Due to the range of file sizes used by the SPECweb99 file set, a significant number of files are less than the size of a page, and hence, are affected. The situation is exacerbated by the Zipf distribution, which results in smaller files being requested more frequently. In fact, more than 50% (see Table 3.1) of requests are for files of a size that occupy one page or less. The result is a significant amount of unnecessary disk accesses for the workloads in this chapter. In order to alleviate this problem, I helped to devise a patch for the 2.6.16-18 Linux kernel to circumvent this behaviour (see Section A.1 for patch). This patch has been approved and is included in newer versions of the Linux kernel. This kernel problem also means that the results presented by Pariag *et al.* [45] are not directly comparable.

The clients, server, network interfaces and switches have been sufficiently provisioned to ensure that the network and clients are not the bottleneck. In particular, the server and client machines require multiple gigabit Ethernet interfaces. To take advantage of multiple Ethernet cards, separate subnets are used for each interface, allowing for explicit load balancing of requests. Four subnets are used to connect the server and client machines via multiple 24-port gigabit switches. Each machine has 4 one-gigabit network interfaces and these interfaces are connected to the four subnets with no machine having multiple interfaces connected to the same subnet. Each copy of the workload generator on a client machine uses a different subnet and simulates multiple users who are sending requests to and getting responses from the web server. Though each client machine only uses two subnets, the clients are equally spread over the four interfaces available on the server, see Figure 3.2. For example, client 1 generates requests on subnets 1 and 2, while client 2 generates requests on subnets 3 and 4.

One limitation of this network configuration is that the clients and server all communicate using fast, reliable network links. A more realistic environment would include a mixture of link speeds from slow to fast, similar to actual clients. Slow network links limit throughput to certain clients, resulting in increased transmission times and additional TCP overheads. Since these conditions where not tested, their effect on the server architectures examined is unknown; however, it is likely the performance of the servers would change. Dealing with these changes could range from requiring additional connections or a higher level of concurrency, especially for the servers using blocking sendfile, and may not be consistent across the server architectures. As well, the overall throughput of the servers would probably be lower. Nevertheless, I conjecture that many of the performance trade offs among the various server architectures would remain the same in this environment as the environment used in this thesis.

3.6 Cache Warming

The experiments were run without clearing the file-system cache between rates and experiments. As well, if the file-system cache is not warmed, a preliminary experiment is run to warm the cache before the beginning the actual experiments. This approach is reasonable as the idea is to measure performance of the web server after initialization and when a working set has been created in the file system. Hence, the experiments are not completely independent as one run affects the immediately following run based on what is left in the file-system cache. However, the state of the file-system cache should be acceptable given that the log files used by the clients never change and that the requests follow a Zipf distribution.

However, this strategy is biased against knot-cache as its application cache is not warmed, and therefore, starts empty after each rate. Hence, there is a penalty to warm up its application cache as the experiment progresses, but it can still take advantage of the warmed file-system cache. Some experiments were run to measure the extent of the bias by warming knot's cache (see Section 3.10.1.1). The other approach is to zero the file-system cache, but that seems inappropriate.

3.7 Table Calculation

Most of the results of the experiments conducted in this thesis are presented in tables. A table entry is an ✕ if the experiment failed to verify. If the experiment verified, the performance of the multiple experiment runs at different request rates is condensed into a single number. These numbers can be compared to determine how experiments perform relative to each other, with larger values indicating better performance. Ideally, it would be better to present the results of each experiment in a graph. However, as the number of experiments is large and the number of lines that can reasonably be placed on a single graph is small, using graphs is impractical for most cases. In addition to the number of graphs required to show each experiment, comparing performance among graphs is also difficult. For these reasons, a single value is used as a coarse representation of the performance of an experiment. Graphs are selectively included in order to examine certain experiments in more detail.

The reason experiments composed of multiple runs at different rates can be condensed is that most of the graphs have a very consistent shape (see Figure 3.4, page 50). The general shape results from the following behaviour. For most of the experiments, the lower request rates can be fully serviced resulting in increasing throughput as the request rate is increased. Eventually, the server reaches a peak, typically around 15,000 requests per second. After this point, the server is saturated and has relatively flat performance with a slight decrease in throughput from peak. Experiments where this assumption does not hold true are marked in red; these anomalies are explained further.

Given the common shape of a curve for an experiment, it is mostly the height of the throughput peak that differentiates each curve. By taking the area under the curve, a single value can be used to represent the performance of an experiment. This area is then normalized to generate a smaller value, because working with ten digit values is cumbersome. Since the lowest and highest request rates are consistent for all the graphs, the area is a reasonable measure as experiments that have higher throughput have larger areas compared to experiments with lower throughput. Hence, larger values represent better performance.

It is possible to normalize in a number of different ways. For example, dividing by the width of the endpoints of the x-axis results in average throughput for the experiment. Unfortunately, throughput is limited by the request rates below peak, meaning that the average throughput can be misleading as it is lower than the throughput at peak and after saturation. Hence, I chose to normalize using an arbitrary value so no additional information aside from relative performance can be inferred from the value. The value chosen was to divide the area by 13 million to give smaller, unitless performance values.

More formally, let the request rates for the experiment be represented by $rate_i$, with $rate_i < rate_{i+1}, \forall i$ and let the corresponding throughput be represented by $tput_i$. In this case, $tput_i$ is actually goodput, where goodput is the throughput of the requests completed within the 10 second timeout. Partially completed requests that timeout are not included in this calculation. Then the condensed area of an experiment consisting of n rates ($n > 1$) can be represented by:

$$Condensed\ Area = (\sum_{i=1}^{n-1}(rate_{i+1} - rate_i) \times (tput_{i+1} + tput_i)/2)/scaling factor$$

As mentioned, for this thesis $scaling factor = 13,000,000$.

3.8 Servers

There are three general servers compared in this chapter. Knot is a thread-per-connection server, *μ*server is an event-driven server and WatPipe is a pipeline server. In order to perform a fair comparison and to highlight differences in architecture, every effort was made to eliminate implementational bias where possible and the servers are made as consistent as possible.

- All the servers, except caching Knot (knot-c), use essentially the same code for implementing the cache table of file descriptors and HTTP headers
- Level of compiler optimization is -O2
- All the servers, except knot-c use sendfile
- All the servers used edge-triggered epoll

Minor differences include using C for *μ*server and C++ for the other servers.

3.8.1 Knot and Capriccio

Previous studies [44, 59] have shown that for various workloads thread-per-connection servers are unsuitable for high performance web servers. In order for a server to perform well under high loads, it must support a large number of simultaneous connections [9], meaning a large number of concurrently running threads for a thread-per-connection server. Unfortunately, as the number of threads increases, overheads related to scheduling, mutual exclusion and synchronization also increase and tend to inhibit performance. However, von Behren *et al.* [57] have suggested the problem is not with the thread-per-connection architecture but with the implementation of the underlying threading libraries. Specifically, any threading library that uses a 1:1 threading model does not scale well when thousands of concurrent threads are required. With a 1:1 model, each user-level thread corresponds to an underlying kernel thread, so overheads related to context switching, contention and scheduling eventually cause the performance of the server to degrade as more threads are added.

Capriccio [57] is a user-level threading package designed to work well for high-concurrency applications. It is designed to efficiently support a large number of concurrent user-level threads by using a non-preemptive, M:1 threading model. With an M:1 threading model, all the user-level threads are multiplexed over a single kernel-thread. This type of threading, combined with no preemption, reduces contention because little or no locking is required since the user-level threads only give up control at fixed scheduling points. Essentially, Capriccio implements threading by adding a scheduler and I/O facilities to a coroutine library [18]. Building non-preemptive, M:1 threading on top of coroutines is similar to co-operative task management with automatic stack management as described by Adya *et al.* [3]. While this approach has the advantage of being fast with low overhead, it cannot take advantage of multiple CPUs, where locking must occur.

Since Capriccio only has a single kernel-thread, allowing user threads to directly make I/O system-calls is problematic. These I/O calls could block the underlying kernel thread essentially causing the entire program to block. Instead, Capriccio has wrappers for I/O system-calls that interact with the scheduler thread and an I/O subsystem to transparently prevent the kernel thread from blocking unnecessarily. Socket and disk I/O each have a specific wrapper implementation.

For socket I/O, the wrapper begins by allowing the user thread to attempt the system call in non-blocking mode. If the call completes successfully, the wrapper returns and the user thread continues execution. However, if the operating system indicates the call would block (EWOULDBLOCK), then an I/O request structure is created, the associated socket descriptor is added to the interest set for the appropriate event mechanism and the user thread blocks. Periodically or when there are no other threads to schedule, the scheduler thread polls the event mechanism to determine if any sockets are ready for reading or writing. Based on the set of ready events, the scheduler retries the associated system calls on behalf of the blocked user threads. If the subsequent system call completes without the system indicating the call would block, then the associated user thread is placed back on the ready queue and the socket descriptor is removed from the interest set. If the I/O request cannot be completed without blocking, for example, the amount of data being transmitted is larger than the available socket buffer space, then the scheduler completes the request using multiple non-blocking system calls with the parameters appropriately adjusted for each call. Similar to the initial request, the additional system calls are performed when subsequent polling indicates the socket descriptor is ready again.

For disk I/O, the wrapper passes the request to a pool of kernel threads referred to as workers. These worker tasks are necessary because disk I/O is potentially blocking if the data is not in the file-system cache and non-blocking disk I/O is unavailable. These worker tasks spin polling a queue of Capriccio disk-I/O requests and performing the potentially blocking disk-I/O on behalf of the user-level threads. When a user thread invokes a disk-I/O wrapper, the user thread creates an I/O request structure, places it on the worker queue for processing and blocks. A worker task removes the queued request, performs the associated system call on behalf of the user thread, and places the user thread back on the ready queue once the call is completed. Safe access to the disk-I/O queue by multiple threads requires appropriate locking.

Knot [57] is a thread-per-connection web server built using the Capriccio threading library. Knot can be run in different modes depending on various compilation and command-line parameters. At compile time, a fixed number of worker tasks is specified and the event mechanism, either poll or epoll, is chosen.

The user threads in Knot can either be pre-forked during server initialization or created on demand for new connections. Previous research [13, 56] reports that statically pre-forking threads results in better performance. With pre-forked user threads, each thread executes a continuous loop accepting connections; with on-demand threads, a single thread accepts connections and then dynamically creates additional threads to process the requests. Once a connection is accepted, the associated thread reads an HTTP request and completely processes the request before reading the next request from the same client.

Knot uses an application-level cache that stores HTTP headers and file data. If a file is not in cache, a cache entry is created consisting of an HTTP header and the file data, which is read from disk. All requests are serviced from the appropriate cache entry using the write system call to send the data from the cache to a client.

3.8.1.1 Modifications to Capriccio and Knot

To allow for a fair comparison, several modifications were made to both Knot and Capriccio. These changes serve to make all the servers consistent where possible and to add features or to fix Knot and Capriccio so that they are implemented using best practices.

The first major change is the addition of sendfile support to Capriccio. After Capriccio supports sendfile, Knot is modified to optionally operate using sendfile instead of write. Previously supported I/O system-calls in Capriccio can be classified as either involving socket I/O or disk I/O. sendfile is an interesting system call as it can involve both socket and disk I/O. Given that these two types of I/O are handled separately in Capriccio, supporting sendfile in Knot required a different implementation than previously supported system calls. Three sendfile variations were implemented; two non-blocking versions and a blocking version. Similar to other I/O functions in Knot, sendfile has a wrapper.

Initially, non-blocking sendfile was implemented in Capriccio as a socket request followed by a disk request. The socket portion involves checking if the socket is ready for writing and the disk portion involves writing file data to the socket with the sendfile system call. In detail, the socket is already in non-blocking mode so the sendfile wrapper begins by creating a socket-I/O request data-structure, adding the socket descriptor to the interest set for the event mechanism and blocking the user thread. When the operating system reports that the socket is writable, however, the sendfile system call is not immediately invoked by the scheduler thread; instead, the user thread is put back on the ready queue. Once the user thread is rescheduled, it wakes up in the wrapper routine, creates a disk-I/O request data-structure, adds it to the disk-I/O queue and blocks again. Eventually one of the worker tasks retrieves the request from the queue and performs the sendfile call and then puts the user thread back on the ready queue. As the socket is in non-blocking mode, the entire file may not be written by a single call to sendfile. When the user thread restarts, it may need to perform additional sendfile wrapper calls to completely transfer the file.

Unfortunately, when this version was tested, it exhibited poor performance due to high event-polling and context-switching overheads. For all other socket I/O operations in Capriccio, the I/O operation is first attempted and only if the operating system indicates that the call would block is the event mechanism used. This technique tends to keep the event mechanism overhead low as the first attempt is usually successful. However, since sendfile may require disk I/O, it is undesirable for a user thread to make the call directly.

To alleviate this problem, a second version of non-blocking sendfile was implemented. This implementation is consistent with the idea of attempting the system call first and then using the event mechanism only if necessary. The sendfile wrapper begins with the user thread creating a disk-I/O request data-structure, adding the request to the disk-I/O queue and blocking the user thread. A worker task removes the request from the queue and attempts the sendfile call with the socket in non-blocking mode. If the operating system indicates that the call would block, then the worker task calls poll directly on that socket descriptor with a one second timeout. If the poll call indicates the socket is writable within the timeout period, the sendfile call is retried. Otherwise, the poll call times out and an appropriate return code is passed back to the user thread. The downside of this approach is that it can cause the worker tasks to block on poll, but with sufficient worker tasks this is not problematic.

The implementation of the blocking sendfile version is similar to the second non-blocking implementation. The sendfile wrapper creates a disk-I/O request data-structure, places the request on the disk-I/O queue and blocks the user thread. A worker task removes the request from the queue, places the socket in blocking mode and attempts the sendfile call. Once the call completes, the user thread is put back on the ready queue. There are a couple of differences between the blocking sendfile version and the second non-blocking sendfile version. First, the blocking version does not need to call poll as the system call now blocks if necessary. Second, a blocking sendfile call always transmits the entire file and so the overhead of waking the user thread to repeatedly call sendfile is avoided. Both the second non-blocking sendfile and the blocking sendfile implementations are used for the experiments in the thesis.

The second major change is to the cache in Knot. For all the versions of Knot tested, the hashing algorithm is changed to be consistent with the hashing algorithm used by *μ*server and WatPipe. This hashing algorithm performs better on the URLs for the file set used by these experiments. As well, for the versions of Knot using sendfile, further cache changes were required. Since sendfile uses the file-system cache, the application cache in Knot is inappropriate for the versions using sendfile. For these versions of Knot, the caching code was modified to be similar to *μ*server so that only file descriptors and HTTP headers are cached.

The third major change is the addition of code to perform cache warming in Knot. The cache-warming code allows a set of files to be read into the application cache when the server starts so the cache is warmed before the experiment begins. In the case of knot-c, the file data is also read into the application cache.

Finally, various bugs were fixed throughout Capriccio and Knot. These bug fixes include problems with epoll support, the I/O subsystem and scheduling.

The first non-blocking sendfile implementation, the blocking sendfile implementation and the cache changes (not including cache warming) were made by other members of the project [45]. The remaining changes and fixes were done as part of the work for this thesis.

3.8.2 μserver

In the Single-Process Event-Driven (*SPED*) architecture, a single thread services multiple connections in various stages of processing using non-blocking I/O. Specifically, a SPED server maintains a set of connections that are being processed and data structures to encode the current status of each connection. An event mechanism such as select is called to determine which connections have outstanding events. For each outstanding event, an appropriate event handler is invoked to process that event.

The μserver originated as an event-driven (SPED) server and has many configuration options, including using either select, poll or epoll as its event mechanism. It also supports zero-copy sendfile and only caches HTTP headers and open file descriptors. The major problem with a SPED server is blocking disk I/O; when the single SPED process blocks for disk I/O, the entire process blocks, hence there is no way to overlap CPU execution and I/O. Depending on the amount of disk I/O required, this can result in the server spending a significant amount of time blocked waiting for disk I/O. As well, a single process cannot take advantage of multiple CPUs. μserver evolved to support two different architectures to deal with these problems. It can run as either a Symmetric Multi-Process Event-Driven (SYMPED) server or a shared Symmetric Multi-Process Event-Driven (shared-SYMPED) server. Note that when running with a single process, both SYMPED and shared-SYMPED revert to a SPED server.

3.8.3 SYMPED Architecture

In SYMPED mode, μserver consists of multiple independent 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 multiple SPED servers to be used in environments where a single copy of the server blocks due to disk I/O. In μserver, the SPED processes are entirely independent except they share a common listening socket. Port sharing is accomplished by having one process create the additional copies of the server after the listen socket has been initialized. This approach has the advantage of not requiring any user-level mutual exclusion or synchronization. In addition, the common listening socket means that no additional port demultiplexing or load balancing is required.

SYMPED is similar to SMPED by Ren and Wang [37], however, the two architectures were developed independently. The SYMPED model is an extension of the *N*-copy approach described by Zeldovich *et al.* [62]. With N-copy, however, each copy of the web server is started using a different TCP port number, so some method for load balancing across the servers is required. Furthermore, in the presence of disk I/O, it may be beneficial to actually have more copies of the server running than CPUs, so N is no longer equal to the number of CPUs.

3.8.4 Shared-SYMPED Architecture

The main drawback of the SYMPED architecture is that each process executes in its own independent address-space. The price of this independence is that each copy of the server maintains an independent cache of open file descriptors and HTTP headers, resulting in memory duplication. This duplication can produce significant memory overhead. For example, 25,000 open file descriptors in each of 100 SPED processes results in a cache of over 2,500,000 file descriptors and their associated HTTP headers across the entire server. As well, the operating system is required to support an equivalent number of open files plus an additional amount of open sockets. These resource requirements can potentially stress the operating system and significantly increase the memory footprint of the server.

In order to mitigate this problem, the shared-SYMPED architecture was developed. In shared-SYMPED, the processes are independent except for a shared cache of open file-descriptors and HTTP headers. Two changes were required to implement shared-SYMPED in *μ*server. First, the shared-SYMPED processes are created to be independent except that file descriptors are shared among the processes. Second, the area of memory for the cache table is mmaped and shared across all the shared-SYMPED processes. Mutual exclusion is handled using a single futex lock [25] around the entire cache table. This approach significantly reduces the number of open file descriptors in the system, the size of the cache table across the server and the memory footprint of the server. The downside of this approach is that the server processes are no longer independent and multi-threaded program considerations must be addressed, i.e., contention on the cache-table lock.

3.8.5 WatPipe

WatPipe is a web server I implemented using a pipeline architecture. In WatPipe, each stage handles a portion of the processing of an HTTP request. It is implemented in C++ and built from the *μ*server source, so much of the code base is the same or similar; however, the components are restructured into a pipeline architecture. Additional code was added to support threads for each stage, communication using queues, specialized event handling for various stages and mutual exclusion and synchronization where necessary.

While SEDA is designed to allow for the creation of well-conditioned servers via dynamic resource controllers, WatPipe eliminates these controllers to simplify the design while still achieving good performance. One of the primary design goals for WatPipe is to keep overhead low; in addition to eliminating dynamic resource controllers, WatPipe also uses a short pipeline with only a small number of threads in each stage. Keeping the number of threads small allows WatPipe to be built on top of a 1:1 threading library. Wrapping of system calls is unnecessary as each thread can invoke (blocking) system calls directly without causing the entire application to block. In the implementation for this thesis, Pthreads are used to create multiple threads within the same address space. Communication is handled using explicit queues that are used to pass socket descriptors between stages. WatPipe's careful batching of events and shortened pipeline should prevent excessive context switching. Like μserver, WatPipe takes advantage of zero-copy sendfile and uses the same code as μserver to cache HTTP reply-headers and open-file descriptors. Both blocking and non-blocking sendfile are supported. Finally, either epoll or select can be used to wait for events. In contrast, SEDA and Haboob are implemented in Java. Haboob has a longer pipeline and utilizes dynamic resource controllers to perform admission control on overloaded stages.

Specifically, the WatPipe implementation consists of 5 stages: Accept, Read Poll, Read, Write Poll and Write. The pipeline server in Figure 2.4 on page 11 is actually the WatPipe server. The first 4 stages have one thread each, simplifying these stages as there is no concurrency within a stage, and stage 5 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 (Accept) accepts connections and passes newly accepted connections to stage 3. Stage 2 (Read Poll) uses an event mechanism to determine which active connections can be read and passes these events to stage 3. Stage 3 (Read) performs reads on these connections, parses the incoming HTTP requests and if necessary opens the required files and adds the appropriate information to the application cache. Stage 4 (Write Poll) uses an event mechanism to determine which connections are available for writing. Once stage 4 determines the connections that can be written, the threads in stage 5 (Write) perform the actual writes. In the case where non-blocking sendfile is used, a request may cycle within stage 5 until all bytes are written. After all the data is written, the connection is passed back to stage 3 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. WatPipe also allows multiple threads in the Read stage, but they were unnecessary for the experiments in this chapter. The dashed lines to Read Poll and Write Poll indicate that communication between the stages occurs implicitly, as the read and write interest sets are maintained in the kernel, so no direct communication into these stages is required.

3.9 Static Uniprocessor Workloads

The next few sections contain the experiments run for each server on the various workloads. The workloads are generated by keeping the file set and client log files consistent and adjusting the memory size of the server machine. The three workloads are labelled based on the size of the server memory: 1.4 GB, 4 GB and .75 GB. These workloads correspond to moderate disk I/O, no disk I/O (in-memory) and heavy disk I/O.

For each workload, a number of experiments are run for each server. At the end of each experiment, the throughput of the server is calculated based on output from the clients. Each client tracks the status of each attempted request and the amount of data transferred.

Additional data is gathered in two ways. First, vmstat is run with a five second interval on the server machine. Second, each server tracks various statistics (server statistics), some of which are printed at 5 second intervals throughout the experiment and the remainder of which are printed out as summary data when the experiment terminates. Due to the large amount of data gathered, only the condensed throughput value for each experiment is reported. The remainder of the data is not included in the thesis. However, a summary of some of the data gathered is presented when necessary to provide further explanations. In particular, file-system cache-size, the percentage of time spent waiting for disk I/O (I/O wait), idle time and context-switching information is provided from vmstat.

3.10 1.4 GB

For the experiments in this section, the server machine is configured with 1.4 GB of memory. Since the size of file set is 2.1 GB, the entire file set does not fit completely into the file-system cache. However, given the Zipf distribution of requests, the majority of requests should be serviced from the file-system cache. While the remainder of requests need to be serviced from disk, only a moderate amount of disk I/O should be required.

3.10.1 Tuning Knot

Experiments were run to tune the three versions of the Knot server: knot-c, knot-nb and knot-b. Knot-c is the knot server running with an application cache, knot-nb is the knot server running with non-blocking sendfile and knot-b is the Knot server running with blocking sendfile. For all the Knot servers, the parameters tuned are the number of threads and the number of worker tasks. As Knot runs using a thread-per-connection model, the number of threads corresponds to the maximum number of simultaneous connections the server can handle. The number of worker tasks determines how many blocking disk-I/O operations can occur simultaneously. For knot-c, one further tuning parameter is the size of its application cache.

3.10.1.1 Knot-c

Table 3.2 shows the results of the coarse-grained tuning for knot-c after verification. Each row in the table represents a different number of workers from 1 to 150. The columns are separated into three sections with the results for 10,000 threads in section one, 15,000 threads in section two and 20,000 threads in section three. In each section, the columns represent a different application cache size value from 10 to 700 MB.

A cache size of 10 MB is misleading as the actual size of the cache can be larger. It is at least as big as the cumulative size of the active files being sent. Once the cache size limit is reached, files selected for eviction are marked for removal and the cache size is updated but these files are only deleted from memory once the file has been completely sent. Using a small cache size, such as 10 MB, is interesting because it maximizes the amount of memory available for the file-system cache as effectively only active files are in the application cache.

The term stability is used in the thesis to describe the performance of a web server as the tuning parameters are adjusted. In this context, a server's performance is stable once the condensed area of the server levels off as one or both of the tuning parameters are adjusted. The range of tuning values over which performance is stable is different for the various servers. Based on extensive tuning, the performance of the servers follows a relatively consistent pattern as the tuning parameters are increased, though individual performance varies from server to server. Once the performance of a server stabilizes or begins decreasing across both tuning parameters, the server has reached its best performing configuration and so increasing the tuning parameters further does not result in higher performance.

The best performing configuration in each Table is highlighted in **bold**. Note that throughout the thesis if more than one configuration results in the best performance, the least resource intensive configuration is always chosen as the best for any experiment. Once a sufficient number of workers are present, performance stabilizes, which is reasonable as the incremental cost of adding additional worker tasks is low, i.e., a small memory cost and the overhead of an additional worker task polling for disk operations. Eventually, these additional overheads will degrade performance. The experiments show the best performance for knot-c is around 10,000 threads, 400 MB of application cache and at least 25 workers. Based on this general vicinity, additional experiments were run and are presented in Table 3.3. For this table, there are two sections with 10,000 threads in section one and 13,000 threads in section two. The columns are varying application cache sizes from 300 to 500 MB. Each row represents a different number of workers from 10 to 150. This table shows that the best performance occurs with 10,000 threads, 400 MB of application

	Cache size in MB											
	10,000 threads				15,000 threads				20,000 threads			
Workers	10	100	400	700	10	100	400	700	10	100	400	700
1	1.48	1.65	1.63	✕	1.47	1.64	1.30	✕	1.46	1.62	✕	✕
5	1.48	1.67	1.61	✕	1.47	1.64	1.26	✕	1.46	1.62	✕	✕
25	1.48	1.70	1.77	1.33	1.47	1.67	1.70	✕	1.46	1.64	✕	✕
50	1.48	1.69	1.78	1.60	1.47	1.67	1.73	✕	1.46	1.64	1.71	✕
100	1.48	1.68	**1.79**	1.72	1.46	1.66	1.75	✕	1.46	1.64	1.72	✕
150	1.47	1.69	1.78	1.70	1.46	1.66	1.76	✕	✕	✕	✕	✕

Table 3.2: Knot cache initial experiments - 1.4 GB (condensed area)

	Cache size in MB					
	10,000 threads			13,000 threads		
Workers	300	400	500	300	400	500
10	1.74	1.72	1.43	1.72	1.66	✕
25	1.76	1.77	1.70	1.74	1.73	1.58
50	1.77	1.77	1.76	1.77	1.75	1.71
100	1.77	**1.78**	1.76	1.76	1.76	1.73
150	1.77	1.78	1.76	1.76	1.76	1.73

Table 3.3: Knot cache fine tune experiments - 1.4 GB (condensed area)

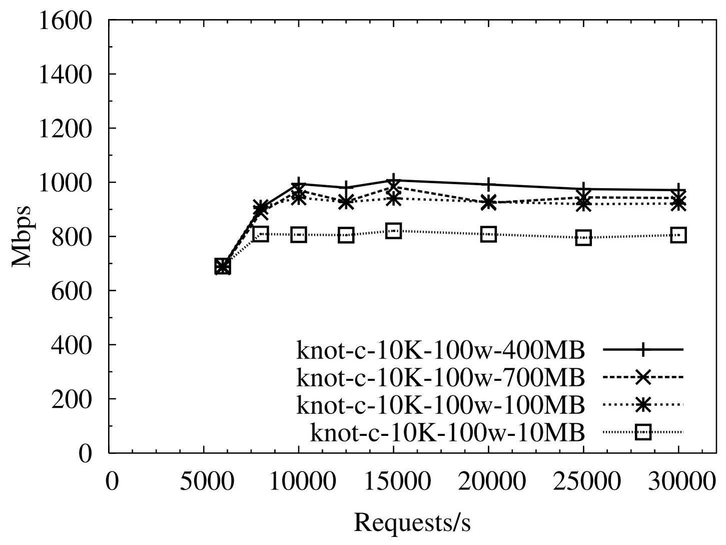


Figure 3.3: Knot cache performance with various cache sizes

cache and 100 workers (1.78), with a peak throughput of 1002 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 956 Mbps at 30,000 requests per second.

For knot-c, 10,000 threads (maximum simultaneous connections) gives the best performance. As is shown later, only supporting 10,000 simultaneous connections is a bottleneck that throttles server performance with this workload so no experiments are shown with values less than 10,000 connections. Hence, the performance of knot-c is capped lower than the other servers since its best performance occurs with 10,000 threads. A significant problem appears to be the amount of memory consumed by each thread. While the stack size of each thread is only 16 KB, adding 5000 threads can consume up to an additional 78 MB of memory just for the stacks. For example, with 100 workers and 400 MB of application cache, the average size of the file-system cache is about 795 MB for 10,000 threads, 709 MB for 15,000 threads and 623 MB for 20,000 threads. Hence, when compared to 10,000 threads, the file-system cache is approximately 11% smaller with 15,000 threads and 22% smaller with 20,000 threads. This problem is more pronounced with knot-c than with the other versions of knot as knot-c's memory footprint is already larger due to its use of an application cache.

The focus of discussion for knot-c is on application cache size as the other Knot configurations use `sendfile`, which uses the file-system cache to store the contents of the files. Figure 3.3 shows how the throughput of knot-c changes as the size of the application cache is varied. Client request rate is plotted on the horizontal axis and the corresponding server throughput in megabits per second is plotted on the vertical axis. All the experiments shown are for 10,000 threads (10K) and 100 workers (100w) with cache sizes of 10, 100, 400 and 700 MB. Up to 400 MB, increasing the size of the application cache improves performance. Unexpectedly, however, increasing the cache size beyond 400 MB causes performance to degrade and for 15,000 threads or more, experiments stopped verifying (see Table 3.2). As long as free memory exists, a larger application cache size should benefit the server, meaning that the performance of Knot should improve up to an application cache-size of around 1.2 GB based on memory information gathered from vmstat during the Knot experiments.

The experiments in this chapter are all run with a warmed file-system cache. However, this method represents a bias against the knot-c experiments as it also uses an application cache to store file data, but this application cache is not warmed since the server is restarted between runs for each request rate in a particular configuration. Furthermore, there is a tension between the application cache and the file-system cache. Having two caches means duplication, resulting in wasted space. Ideally, the application cache size should be as large as possible in order to minimize overheads when servicing requests. As there is not enough memory to hold the entire file set in the application cache, overheads occur such as copying file data from the file-system cache to the program memory every time a file is inserted into the application cache and the additional management of the application cache as entries are inserted or evicted. However, as shown in the experiments, an application cache larger than a certain size can cause performance to degrade and experiments not to verify. This result is somewhat counter-intuitive because a larger application cache size should benefit the server as it can service more requests from its own cache. Unfortunately, a larger application cache means that the size of the (warmed) file-system cache is smaller when the server initially starts. Hence, at the beginning of the run, more requests are sent to disk, resulting in timeouts early in the experiment that result in the experiment not verifying.

In order to overcome this bias, some knot-c experiments were run with 100 workers but without restarting the server between successive request rates of the same experiment. Keeping the server running between rates means that the application cache is warm for all but the first run at the lowest rate. Without performing a full tuning, it is difficult to judge the complete performance picture; however, these experiments do provide some additional insights. Experiments with this cache warming strategy (not shown) were run with 400 MB, 700 MB and 1 GB of application cache and 10,000 and 15,000 threads. Interestingly, the performance with 400 MB and 700 MB for 10,000 threads is about the same (1.80) but the performance with 1 GB is lower (1.72). The performance of the 400 and 700 MB experiments appear to be capped by the 10,000 threads. With 1 GB of application cache, the file-system cache size was about 145 MB. It appears that given the presence of disk I/O, a certain amount of file-system cache is needed in order to get reasonable performance. Hence, a file-system cache of 145 MB appears to be too small. At 15,000 threads and 100 workers, the performance with 400 MB stays about the same (1.81) but the performance with 700 MB of cache increases by 4% (1.87) and with 1GB of cache the experiment did not pass verification. These experiments show that running knot-c without a warmed application cache results in only a small decrease in performance for this workload, so the previous experiments without cache warming sufficiently characterize knot-c's performance.

The big drawback of knot-c is the need for an application cache. First, requiring two caches with duplicated data means the effective cache size for knot-c is always smaller than for the other servers without an application cache. Second, the application cache for knot-c is not warmed between runs, resulting in performance and verification problems. Switching to sendfile allows the application cache for file data to be eliminated and the performance of knot to be evaluated on a more equal footing with the other servers.

3.10.1.2 Knot-nb and knot-b

Table 3.4 shows the results of the coarse-grained tuning for knot-nb and knot-b after verification. Each row in the table represents a different number of workers from 1 to 150. The columns are separated into two sections with the results for knot-nb in the first section and the results for knot-b in the second section. In each section, the columns represent the number of threads.

Consider the knot-nb section (left) in Table 3.4 first. Similar to knot-c, all the experiments with 20,000 threads did not verify. However, in this case the area of best performance occurs with 15,000 threads and at least 5 workers. With non-blocking sendfile, fewer workers are required as there is less copying between the kernel and user-space than knot-c. Based on this general vicinity, additional experiments were run and are presented in Table 3.5(a). For this table, the number of workers are varied from 15 to 150 and the number of threads from 13,000 to 17,000. This table shows that the best performance occurs with at least 15 workers. As the performance is similar for both 13,000 or 15,000 threads, the least resource intensive configuration is chosen. In this case, the best performance occurs with 13,000 threads and 15 workers (2.18), with a peak throughput of 1280 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1240 Mbps at 30,000 requests per second.

Again, once a sufficient number of workers are present, performance stabilizes. Unfortunately, the workers in Capriccio poll, even when there are no requests to process, so idle time and I/O wait are almost zero. Hence, it is impossible to use these values to tune the number of workers.

As can be seen in Tables 3.4 and 3.5(a), the number of threads is an important tuning parameter. Too few threads hinder performance and too many threads result in verification failures. With 10,000 or fewer threads, throughput never exceeds 1030 Mbps. In this case, performance for knot-nb is limited because it cannot support a sufficiently large number of simultaneous connections. In all the experiments with a memory size of 1.4 GB, when the number of simultaneous connections is capped at 10,000, throughput never exceeds around 1030 Mbps, and the condensed area is approximately 1.82 for these experiments.

For every server configuration using non-blocking sendfile, there is a point where too many connections result in verification failures. The actual point where experiments begin to fail verification varies from server to server. When the number of threads in knot-nb is larger than 15,000, experiments tend to stop verifying.

As well, the relationship between workers and threads can be seen. With 10,000 threads, performance improves as the number of workers is increased to around 5. After this point, performance does not improve as there are an insufficient number of connections (threads). With 5 workers, increasing the number of threads from 10,000 to 15,000 improves performance by almost 20%. At peak, performance improves from 1023 Mbps to 1259 Mbps, an increase of 23%. Adding additional workers beyond 5 does result in a small improvement in performance, but performance stabilizes once there are at least 5 workers.

Now consider the knot-b (right) section in Table 3.4. Unlike the knot-c and knot-nb case, experiments with 20,000 threads did verify. In this case, the area of best performance occurs with 15,000 threads and 100 workers. More worker tasks (kernel threads) are needed with blocking sendfile because the kernel thread may block for both socket and file operations. Hence, the kernel thread is unavailable and cannot be used by the server for other work, and therefore additional kernel threads are required.

Based on this general vicinity, additional experiments were run and are presented in Table 3.5(b). For

Workers	Number of Threads					
	non-blocking sendfile			blocking sendfile		
	10,000	15,000	20,000	10,000	15,000	20,000
1	1.77	×	×	×	×	×
5	1.81	2.17	×	×	×	×
25	1.82	2.18	×	1.70	1.72	1.48
50	1.82	2.18	×	1.82	2.15	2.09
100	1.82	**2.19**	×	1.83	**2.16**	2.15
150	1.82	2.18	×	1.82	2.16	2.15

Table 3.4: Knot sendfile initial experiments - 1.4 GB

Workers	Number of Threads		
	13,000	15,000	17,000
15	**2.18**	2.18	×
25	2.18	2.18	×
50	2.18	2.18	×
75	2.18	2.18	×
100	2.18	2.17	×
125	2.18	2.18	×
150	2.18	2.18	×

(a) non-blocking sendfile

Workers	Number of Threads		
	13,000	15,000	17,000
35	2.04	2.01	1.96
50	2.16	2.15	2.13
75	**2.17**	2.15	2.14
100	2.16	2.14	2.14
125	2.17	2.14	2.14
150	2.17	2.15	2.14
200	2.15	2.14	2.13

(b) blocking sendfile

Table 3.5: Knot sendfile fine tune experiments - 1.4 GB

this table, the number of workers are varied from 35 to 200 and the number of threads from 13,000 to 17,000. This table shows the best performance occurs with 13,000 threads and 75 workers. Consistent performance from 50 to 200 workers indicates there are sufficient worker tasks. The best performance occurs at 13,000 threads and 75 workers (2.17), with a peak throughput of 1261 Mbps occurring at 15,000 requests per second and a sustained throughput of around 1237 Mbps at 30,000 requests per second.

Similar to knot-nb, throughput with 10,000 threads is capped at 1030 Mbps. As expected, the blocking sendfile version requires more workers to achieve good performance, at least 50 instead of 15 with the non-blocking version. Another interesting feature is that experiments with less than 50 workers tend to exhibit performance degradation after peak. In fact, all the experiments with fewer than 50 workers either did not verify or had tails that dropped by more than 20%, as indicated by the red entries in the tables.

It is interesting to note that the best knot-nb and knot-b configurations have very similar performance for this memory size. However, despite this similarity, there are some notable differences. Unlike knot-nb, knot-b seems to be able to support a larger number of threads (connections) without verification failures. In fact, the blocking sendfile version of all the servers tends to be able to support a larger number of connections without verification failures when compared to their non-blocking counterparts. This verification problem is explored in section 3.10.2. The downside of using blocking sendfile is that for an equivalent number of simultaneous connections, the number of workers required to service those connections is larger resulting in more overhead and decreasing throughput. This additional overhead is more problematic when the system is under increased memory pressure. The benefit of having a kernel thread dedicated to sending the entire file is that as the number of connections increases, experiments still verify despite increased overheads and lower throughput.

3.10.2 Tuning *μ*server

Experiments were run to tune the four versions of *μ*server: symped-nb, symped-b, sharedsymped-nb and sharedsymped-b. These four versions cover the space of non-sharing (symped) versus sharing (sharedsymped) of file descriptors and cache table, and non-blocking (-nb) versus blocking (-b) sendfile. For all versions of *μ*server, the parameters tuned are the maximum number of simultaneous connections and the number of processes (number of copies of the server running). The number of processes determines how many disk I/O operations can be pending. These operations block the kernel thread if the data is accessed from disk or are non-blocking if the data is in the file-system cache. With blocking sendfile, the number of processes also determines the maximum number of files concurrently being sent at any given time.

3.10.2.1 Symped-nb and symped-b

Table 3.6 shows the results of the coarse-grained tuning for symped-nb and symped-b. Each row in the table represents a different number of processes from 1 to 150. The columns are separated into two sections, with the results for symped-nb in section one and the results for symped-b in section two. In each section, the columns represent a different maximum number of connections from 10,000 to 30,000.

The experiments show the best performance for symped-nb is around 25 processes and 15,000 connections (2.21). Based on this general vicinity, additional symped-nb experiments were run and are presented in Table 3.7(a). For this table, the number of processes are varied from 5 to 75 and the maximum number of connections from 13,000 to 17,000. Table 3.7(a) also shows the best performance occurs with 25 processes and 15,000 connections (2.21), with a peak throughput of 1275 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1264 Mbps at 30,000 requests per second.

	Maximum Number of Connections									
	non-blocking sendfile					blocking sendfile				
Procs	10,000	15,000	20,000	25,000	30,000	10,000	15,000	20,000	25,000	30,000
1	1.70	1.67	1.67	1.67	1.65	×	×	×	×	×
5	1.81	×	×	×	×	×	×	×	×	×
25	1.82	**2.21**	×	×	×	1.57	1.70	1.54	1.41	×
50	1.82	2.14	×	×	×	1.75	2.06	2.09	2.04	1.96
100	1.75	×	×	×	×	1.76	2.04	2.09	**2.11**	2.11
150	1.68	×	×	×	×	1.70	1.95	2.03	2.05	2.06

Table 3.6: μserver SYMPED initial experiments - 1.4 GB

	Max Number of Connections		
Procs	13,000	15,000	17,000
5	2.07	×	×
10	2.18	×	×
15	2.18	×	×
25	2.15	**2.21**	2.19
35	2.12	2.19	×
50	2.07	2.15	×
75	2.02	2.08	×

(a) non-blocking sendfile

	Max Number of Connections				
Procs	20,000	25,000	30,000	35,000	40,000
50	2.10	2.05	1.96	1.85	1.77
75	2.12	**2.14**	2.14	2.11	2.07
100	2.09	2.11	2.11	2.12	2.12
125	2.06	2.08	2.09	2.09	2.09
150	2.03	2.06	2.06	2.07	2.06

(b) blocking sendfile

Table 3.7: μserver SYMPED fine tune experiments - 1.4 GB

All the symped-nb experiments with a single process (SPED) verified because they are self limiting. With SPED, a single process performs the tasks of accept, read and write in a continuous cycle. During the accept phase, the process accepts all available connections until the operating system indicates that there are currently no more connections to be accepted or the maximum number of simultaneous connections is reached. Once the maximum number of simultaneous connections is sufficiently large, the size of the pending accept queue and the rate at which new connections are arriving determine how many connections are accepted at any given time. If this rate is roughly balanced by the number of connections that are closed, then a steady state is reached. At this point, further increasing the maximum simultaneous connections parameter does not affect server function but simply increases the memory footprint of the server because the size of the static data structures increases but additional connections are not accepted. It may be possible to force the server to accept more connections, e.g., by increasing the size of the accept queue, but this technique does not improve throughput: since there is no idle time when the server reaches steady-state, the server is already able to accept as many connections as it can handle. Forcing the server to accept additional connections beyond this point results in verification problems as response times increase but throughput does not. For all of the SPED experiments, steady-state occurs with less than 10,800 simultaneous connections on average during an experiment. Increasing the maximum number of connections parameter significantly beyond this value does not benefit the SPED experiments.

All experiments with a maximum number of connections parameter of 10,000 also verified. As with all the servers with 1.4 GB of memory, performance with 10,000 connections is capped at around 1030 Mbps. Comparing all the servers, symped-nb and sharedsymped-nb achieve the best performance for a maximum connections value of 10,000. However, 10,000 connections is too few as all the servers can achieve better performance with a larger maximum-connections parameter.

For experiments involving more than one process, the experiments tend to stop verifying above 15,000 connections. As the maximum connections parameter is increased, the number of file timeouts, especially for large files, increases beyond the verification threshold. A subset of the experiments in Table 3.6 were run again with profiling enabled. Examining the OProfile data (not shown) reveals no spikes or unusual values as the maximum number of connections parameter is increased, even in the costs related to calling the event mechanism. (The stability in the cost of the event mechanism is due to the scalability of epoll.) The reason for the increase in file timeouts seems to be that after the experiment is running for a short time, the number of requests read by the server starts to become larger than the number of requests completed by the server. Despite not being able to keep up with the number of requests in progress, the server continues to read new requests. Because requests for larger files tend to require multiple calls to non-blocking sendfile, these replies tend to timeout before being completely processed. Clearly, the number of simultaneous requests that the server is trying to process is directly related to the number of connections it has accepted, and hence, related to the maximum number of connections parameter.

In fact, the reason for the verification failures with all the non-blocking servers seems to be timeouts on large files. With the non-blocking servers, if a file requires multiple calls to sendfile in order for it to be completely sent, the time between calls to sendfile increases as the number of connections increases. Eventually, the time increases sufficiently to cause the client to timeout before the file is completely transferred. With blocking sendfile, the initial call continues until the file is completely sent, resulting in fewer timeout problems.

To verify the large-file timeout problem, non-blocking sendfile experiments were run for symped-nb with the socket write-buffer set large enough to accommodate the largest files in the file set. Hence, each file can be transmitted with a single call to sendfile. Note that the operating system may still send the file in several chunks, but the server only needs to make a single call to sendfile. If the reason that file timeouts are occurring is that large files require multiple calls to sendfile and these calls get further apart as the

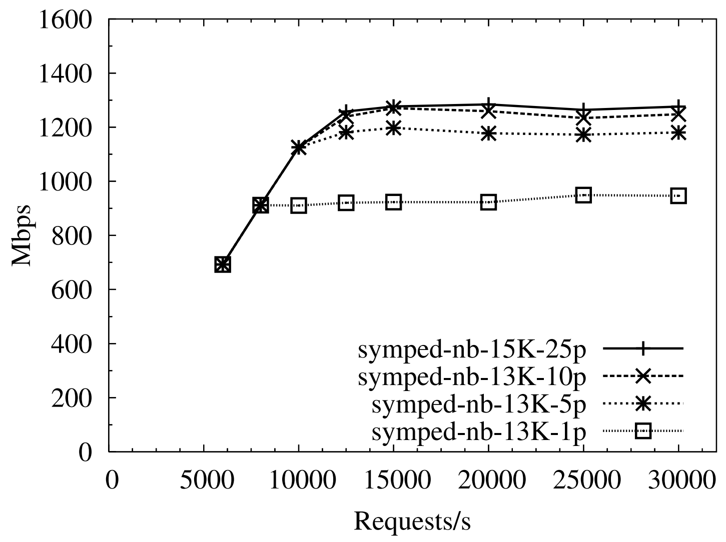


Figure 3.4: μserver with non-blocking sendfile

load increases, then using a large socket buffer should fix this problem. Table 3.8 contains the results of these experiments for 15,000 and 20,000 maximum connections. The main result is that the server is able to handle a larger number of simultaneous connections, further experiments show no verification failures up to 30,000–35,000 maximum connections. When verification failures do start to occur, the distribution of timeouts is across all file sizes and not concentrated on the larger file sizes. The secondary result is a performance boost because the server operates more efficiently without having to perform multiple sendfile calls for a single request. While these are desirable outcomes, further experiments with a large socket buffer are not run as most of the existing file set can already be sent in a single call and it is important to test the server's ability to handle larger files requiring multiple writes. In general, it is impractical to increase the socket-buffer size to accommodate the largest file in the file-set because files can be arbitrarily large, resulting in wasted memory, especially for a large number of simultaneous connections.

Figure 3.4 shows how the throughput of symped-nb changes as the number of connections and processes are varied. The lines labeled 13K show the results for 13,000 connections and one (1p), five (5p) and ten processes (10p) respectively. The line labeled 15K shows the results with 15,000 connections and twenty-five processes (25p).

The throughput of symped-nb-1p-13K is 923 Mbps at 15,000 requests per second and 946 Mbps at 30,000 requests per second. Increasing the number of processes to 5 improves throughput by 29% at 15,000 requests per second and 22% at 30,000 requests per second. Further increasing the number of processes to 10 improves throughput by 6% at 15,000 requests per second and 8% at 30,000 requests per second. At 25 processes, the performance of the server goes down but it has not reached its best

	Max Number of Connections	
Procs	15,000	20,000
5	2.24	2.24
10	2.30	2.27
15	2.30	2.25
25	2.26	2.29
35	2.23	2.30

Table 3.8: μserver non-blocking SYMPED with large socket buffer size

performance (not shown). The problem is that the maximum simultaneous connections value is not large enough and so the additional processes increase overhead without increasing throughput. By increasing both the number of processes to 25 and the number of connections to 15,000, the server achieves its best performance with throughput improving by 1% at 15,000 requests per second and 2% at 30,000 requests per second.

Consider the I/O wait for each of these configurations. With 13,000 connections and 1 process, I/O wait on average is 33% at 15,000 requests per second and 31% at 30,000 requests per second. Increasing to 5 processes results in the I/O wait dropping to 8–9% for both 15,000 and 30,000 requests per second. Moving to 10 processes, the I/O wait drops to 2–3% on average. Finally, with 25 processes, the I/O wait drops to 0. As can be seen in Table 3.7(a), additional processes beyond 25 do not help because the I/O wait has essentially been eliminated. In fact, as additional processes are added, performance drops as more resources are consumed.

While 25 processes drop the I/O wait to 0, performance at 13,000 connections goes down. In this case, 13,000 connections is the bottleneck and the additional processes increase overhead and hurt performance. However, with 15,000 connections the I/O wait is still zero and performance is better as the server is able to handle more connections.

It is important to have both a sufficient number of connections and a sufficient number of processes. When there are too few connections, increasing the number of processes is not beneficial. Furthermore, increasing the number of connections without a sufficient number of processes leads to verification failures. When the number of connections is not the bottleneck, increasing the number of processes improves performance. However, this performance increase is usually because of parallelism afforded by overlapping disk I/O with other activities. Once the I/O wait is zero, additional performance benefits due to concurrency are unlikely.

Table 3.6 also shows the results of the coarse-grained tuning for symped-b. The experiments show the best performance for symped-b is around 100 processes and 25,000 connections (2.11). Based on this

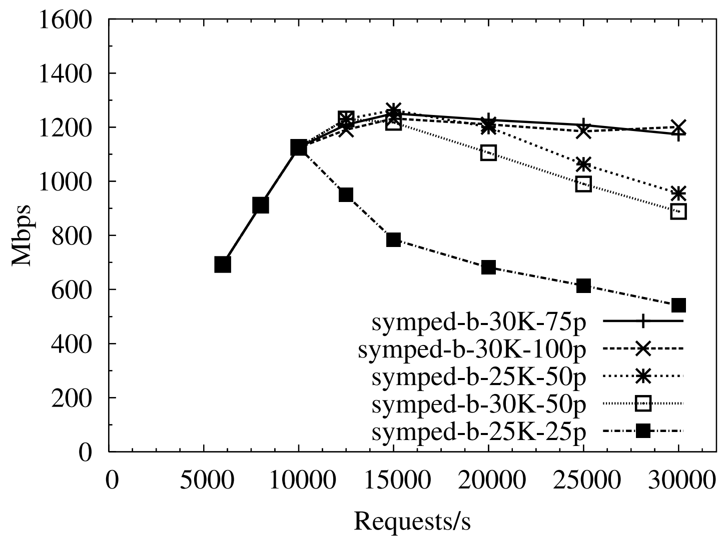


Figure 3.5: μserver with blocking sendfile

general vicinity, additional experiments were run and are presented in Table 3.7(b). For this table, the number of processes are varied from 50 to 150 and the maximum number of connections from 20,000 to 40,000. This table shows that the best performance occurs with 75 processes and 25,000 connections (2.14), with a peak throughput of 1234 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1198 Mbps at 30,000 requests per second.

As expected, blocking sendfile requires significantly more processes than symped-nb. In fact, the symped-b experiments with fewer than 25 processes did not verify and all the experiments with 25 processes had tails that dropped by more than 20% as indicated in red in the table. The experiments with 50 processes and 25,000 or more maximum connections also had tails with large drops. Figure 3.5 shows how the throughput of symped-b changes as the number of connections and processes are varied. The lines labeled with 30K show the results with 30,000 connections and fifty (50p), one hundred (100p) and seventy-five (75p) processes. The lines labeled with 25K show the results with 25,000 connections and twenty-five (25p) and fifty (50p) processes.

The bottom three lines are examples of experiments that are labeled in red in the Table 3.7(b). With these experiments, the throughput increases to a certain point and then drops as the load on the server increases. For all three of these lines, the drop is more than 20%, with smaller table values indicating earlier or steeper drops in performance.

Since symped-b requires additional processes to run well, it has lower performance than symped-nb due to the extra overhead, especially memory. It is interesting to note that symped-b is resistant to large-file timeouts even at high maximum connection values. With non-blocking sendfile, large files are sent in pieces with multiple calls to sendfile. In between sending successive pieces of a large file, many other files or file pieces may get sent. As the number of requests become large and the server gets saturated, the period of time taken to send a large file increases as the processing of more and more requests are interleaved with sending the large file. Eventually, the time it takes to send the file exceeds the time alloted by the client and the request times out. This problem does not occur with small files as they are sent in a single call to sendfile and has less affect with medium files as only a few calls to sendfile are required. With blocking sendfile, the kernel thread performing the call blocks until the entire file is sent, preventing new requests from getting priority over existing requests. In fact, using blocking sockets with sendfile means that files get attention directly proportional to their size.

Unlike *µ*server SPED, with symped-b the number of simultaneous connections does increase as the maximum connections parameter is increased due to the multiple kernel threads. However for rates of 10,000 requests per second or higher, another interesting trend emerges when the number of processes is fixed. Despite an increasing number of simultaneous connections, the number of requests read per second is relatively steady once performance reaches its peak. Note, for the symped-b experiments because of blocking sendfile, the number of requests read is almost equal to the number of replies sent, and hence, is an indicator of throughput. Therefore, the performance of the server also stabilizes once the number of requests read stabilizes. For example, with 100 processes, according to the server output, the number of requests read per second stabilizes with 25,000 or more simultaneous connections. Clearly, this stability would degrade with a large number of connections due to the additional overhead. Hence, the symped-b server is also self limiting but in another way compared to *µ*server SPED. With *µ*server SPED, the number of requests read also stabilizes, but this stability occurs because the number of simultaneous connections stops increasing even as the maximum connections parameter continues to increase.

With 75 processes and a maximum connections parameter of 20,000, the I/O wait is zero at 30,000 requests per second. Increasing the maximum connections parameter to 25,000 results in a small I/O wait value of 1%. With at least 100 processes, the I/O wait is again reduced to zero. However, as the number of processes increases, the amount of overhead, especially memory, increases and the throughput decreases. In fact, the file-system cache shrinks by about 25–30 MB every time the number of processes is increased by 25 because the processes do not share an address space. As the throughput stabilizes with 25,000 or more simultaneous connections, increasing the number of connections does not improve performance. It is possible that a small amount of additional performance exists in the space between 75 and 100 processes.

3.10.2.2 Sharedsymped-nb and sharedsymped-b

Table 3.9 shows the results of the coarse-grained tuning for sharedsymped-nb and sharedsymped-b. Each row in the table represents a different number of processes from 1 to 150. The columns are separated into two sections with the results for sharedsymped-nb in section one and the results for sharedsymped-b in section two. In each section, the columns represent a different maximum number of connections from 10,000 to 30,000.

The experiments show the best performance for sharedsymped-nb is around 25 processes and 15,000 connections (2.27). Based on this general vicinity, additional experiments were run and are presented in Table 3.10(a). For this table, the number of processes were varied from 5 to 75 and the maximum number of connections from 13,000 to 17,000. This table shows that the best performance occurs with 15 processes and 17,000 connections (2.34), with a peak throughput of 1374 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1354 Mbps at 30,000 requests per second.

Similar to symped-nb, all the sharedsymped-nb experiments with a single process (SPED) verified. In fact, running μserver SYMPED or μserver shared-SYMPED with one process is equivalent, except for the cache lock. The difference is the basic overhead of locking as there is no contention in the single process configuration. Comparing the symped-nb and sharedsymped-nb SPED experiments (Table 3.6 versus Table 3.9) shows that the cost of locking is relatively small and that any additional costs associated with a shared cache-table, when there is more than one process, are due to contention. For sharedsymped-nb, the number of simultaneous connections did not exceed 15,000 for all the SPED experiments and on average there were less than 11,000 simultaneous connections in each experiment during the run of the experiment. Therefore, the performance of sharedsymped-nb SPED beyond 15,000 maximum connections is relatively steady.

The sharedsymped-nb experiments involving more than one process tend to stop verifying at or above 20,000 maximum connections. Once the server has at least 15 processes, adding further processes does not improve performance. As more processes are added, performance tends to drop or experiments stop verifying as more resources are consumed and the efficiency of the server degrades. Since the system is under memory pressure, as the memory footprint of the server increases, the size of the file-system cache gets smaller. For example, with a maximum connections value of 15,000 at 30,000 requests per second, the file-system cache is approximately 39 MB smaller (3%) with 75 processes than with 15 processes and 82 MB smaller (7%) with 150 processes. Note that these memory-footprint increases are smaller per process than with symped-nb, showing the advantage of sharedsymped-nb. In addition to the increased memory footprint, the efficiency of the server with respect to calls to epoll_wait also decreases as the number of processes increases. 75 processes make about 2.2 times the number of calls to epoll_wait and 150 processes make about 2.3 times the number of calls to epoll_wait compared to 15 processes. Finally, context switching also increases with the number of processes; 75 processes have approximately 2.9 times more context switches per second and 150 processes about 4.9 times more context switching per second compared to 15 processes. These overheads are enough to lower performance and eventually cause the server to stop performing acceptably based on the verification criteria.

Procs	Maximum Number of Connections									
	non-blocking sendfile					blocking sendfile				
	10,000	15,000	20,000	25,000	30,000	10,000	15,000	20,000	25,000	30,000
1	1.71	1.68	1.67	1.66	1.66	×	×	×	×	×
5	1.81	×	×	×	×	×	×	×	×	×
25	1.83	**2.27**	×	×	×	1.57	1.72	1.56	×	×
50	1.83	2.20	×	×	×	1.75	2.12	2.15	2.10	2.01
100	1.78	2.11	×	×	×	1.79	2.11	2.19	2.21	**2.22**
150	1.72	2.06	×	×	×	1.75	2.05	2.14	2.17	2.19

Table 3.9: *µ*server shared-SYMPED initial experiments - 1.4 GB

Procs	Max Number of Connections		
	13,000	15,000	17,000
5	2.11	×	×
15	2.22	2.31	**2.34**
25	2.20	2.28	2.31
35	2.17	2.25	2.29
50	2.14	2.21	×
75	2.09	2.16	×

(a) non-blocking sendfile

Procs	Max Number of Connections		
	23,000	25,000	27,000
35	1.85	1.78	1.73
50	2.14	2.11	2.07
75	2.22	**2.23**	2.23
100	2.21	2.21	2.22
125	2.18	2.20	2.20
150	2.16	2.17	2.19

(b) blocking sendfile

Table 3.10: *µ*server shared-SYMPED fine tune experiments - 1.4 GB

The expectation is that the performance of sharedsymped-nb should be better than symped-nb for the following reason. Sharedsymped-nb has a smaller memory footprint, making additional memory available for the file-system cache, but has contention for the shared cache-table. Symped-nb has a larger memory footprint but does not have to deal with contention related to sharing a cache table among multiple processes. Assuming that the cache lock is not a bottleneck, sharedsymped-nb should perform better in any situation where there is memory pressure as there is more memory available for the file-system cache.

Comparing the performance of symped-nb and sharedsymped-nb with a memory size of 1.4 GB shows that sharedsymped-nb has a small performance advantage over symped-nb. This performance advantage tends to get larger as the maximum number of connections increases, resulting in more sharedsymped-nb experiments at 15,000 and 17,000 maximum connections verifying. For the best performing configurations of each server, the performance difference is almost 8% at peak. However, the best parameters are different for the two servers. Consider the performance of both servers with 17,000 maximum connections and 25 processes. Symped-nb has a peak of 1288 Mbps and a condensed area of 2.19 (see Table 3.7(a)) and sharedsymped-nb has a peak of 1353 Mbps and a condensed area of 2.31. At 30,000 requests per second, symped-nb's average file-system cache size is 1105 MB and sharedsymped-nb's average file-system cache size is 1200 MB a difference of about 95 MB or 9% larger.

Table 3.9 also shows the results of the coarse-grained tuning for sharedsymped-b. The experiments show the best performance for sharedsymped-b is around 100 processes and 30,000 connections (2.22). Additional experiments run at higher maximum connection values (not shown in table) reveal that performance stabilizes at around 25,000 to 30,000 maximum connections. Based on this stability, additional experiments around 25,000 maximum connections were run and are presented in Table 3.10(b). For this table, the number of processes were varied from 35 to 150 and the maximum number of connections from 23,000 to 27,000. This table shows that the best performance occurs with 75 processes and 25,000 connections (2.23), with a peak throughput of 1295 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1257 Mbps at 30,000 requests per second.

Similar to symped-b, all experiments with fewer than 25 processes did not verify. While experiments with 25 processes did verify, all these experiments had tails that dropped by more than 20% as indicated in red in the table. Hence, 25 processes are insufficient when blocking sendfile is used. Similarly, with 25,000 connections or more, 50 processes are insufficient.

Similar to symped-b, the sharedsymped-b server is self limiting. For rates of 10,000 requests per second or higher, with 100 processes and 25,000 or more simultaneous connections, according to the server statistics the number of requests read per second stabilizes. Despite an increasing number of simultaneous connections, the number of requests read per second is relatively steady once the performance reaches its peak. Again, this stability would degrade once the overhead of an excessively large number of connections uses sufficient additional memory, resulting in a noticeable effect on performance.

Also similar to *μ*server SYMPED, sharedsymped-b has lower performance than sharedsymped-nb. As discussed earlier, the versions of the server using blocking sendfile require more processes to run well, but these additional processes result in increased overheads and affect performance. When using blocking sendfile, there seems to be a tension between using more processes to achieve the required concurrency to run well in the presence of blocking socket operations and the overheads resulting from these additional processes. Given the large number of processes required for the blocking versions of the server, memory is a significant overhead. Due to the sharing of file descriptors and the cache table, the memory footprint of sharedsymped-b is smaller than the memory footprint of symped-b for similar configuration options. As the number of processes gets larger, this memory efficiency begins to become more significant and results in better performance. For example, with 50 processes or more, sharedsymped-b performs better than symped-b. However, unlike with sharedsymped-nb, this improved performance can likely be attributed to a smaller memory footprint.

Comparing the performance of sharedsymped-b and symped-b for 75 processes and 25,000 maximum connections at 30,000 request per second, the throughput of sharedsymped-b is 1257 Mbps versus 1198 Mbps for symped-b. Based on vmstat data, sharedsymped-b has an average file-system cache size of 1109 MB versus 966 MB for symped-b, resulting in a memory difference of around 143 MB or almost 15%. The average I/O wait is 1% for both sharedsymped-b and for symped-b. Interestingly, on average sharedsymped-b has about 7% more context switching than symped-b and about 31% more calls to epoll_wait. The smaller memory footprint of sharedsymped-b is a definite advantage despite the additional overheads resulting from a shared cache-table.

3.10.3 Tuning WatPipe

Experiments were run to tune the two versions of WatPipe: watpipe-nb and watpipe-b. Watpipe-nb is the WatPipe server using non-blocking sendfile and watpipe-b is the WatPipe server using blocking sendfile. For both servers, the parameters tuned are the maximum number of connections and the number of writer tasks. The number of writers determine how many blocking disk I/O operations can occur simultaneously.

Table 3.11 shows the results of the coarse-grained tuning for watpipe-nb and watpipe-b after verification. Each row in the table represents a different number of writer tasks from 1 to 150. The columns are separated into two sections with the results for watpipe-nb in section one and the results for watpipe-b in section two. In each section, the columns represent a different maximum number of connections from 10,000 to 30,000.

The experiments show the best performance for watpipe-nb is around 25 writer tasks and 15,000 maximum connections (2.34). Based on this general vicinity, additional experiments were run and are presented in Table 3.12(a). For this table, the number of writer tasks were varied from 10 to 100 and the maximum number of connections from 13,000 to 17,000. This table also shows that the best performance occurs with 25 writer tasks and 15,000 connections (2.37), with a peak throughput of 1393 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1370 Mbps at 30,000 requests per second.

All watpipe-nb experiments with a maximum number of connections value of less than 25,000 verified. Above 25,000 connections, the experiments tend to not verify. Once the server had at least 25 writer tasks, adding further writers did not improve performance. In fact, the average I/O wait drops to zero once there are at least 25 writers. As more writer tasks are added, performance did not drop appreciably, but eventually the performance would drop as the overhead of adding more writer tasks becomes larger.

For the experiments run, the performance of watpipe-nb is more stable over a larger range for the parameters tested than the other non-blocking servers. While experiments began to stop verifying at

Writers	Maximum Number of Connections									
	non-blocking sendfile					blocking sendfile				
	10,000	15,000	20,000	25,000	30,000	10,000	15,000	20,000	25,000	30,000
1	1.71	1.83	1.75	1.67	1.64	✕	✕	✕	✕	✕
5	1.77	2.22	2.14	2.11	2.10	✕	✕	✕	✕	✕
25	1.77	**2.34**	2.33	✕	✕	1.67	1.79	1.50	1.38	✕
50	1.77	2.33	2.33	✕	✕	1.77	2.28	2.21	2.13	1.98
100	1.77	2.32	2.33	✕	✕	1.78	2.30	**2.31**	2.31	2.31
150	1.78	2.32	2.32	✕	✕	1.77	2.29	2.30	2.31	2.32

Table 3.11: WatPipe initial experiments - 1.4 GB

Writers	Max Number of Connections		
	13,000	15,000	17,000
10	2.23	2.36	2.34
25	2.23	**2.37**	2.37
35	2.23	2.36	2.36
50	2.23	2.36	2.37
75	2.23	2.35	2.36
100	2.22	2.35	2.36

(a) non-blocking sendfile

Writers	Max Number of Connections		
	13,000	15,000	17,000
35	2.13	2.14	✕
50	2.23	2.31	2.29
75	2.23	**2.32**	2.32
100	2.22	2.31	2.32
125	2.22	2.31	2.31
150	2.22	2.31	2.31

(b) blocking sendfile

Table 3.12: WatPipe fine tune experiments - 1.4 GB

25,000 maximum connections, for less than 25,000 maximum connections, performance stabilizes once there are a sufficient number of writer tasks. As the entire address space is shared, the cost of an additional writer task is mainly its stack. More importantly, the remainder of the server functions the same, so adding more writers should not affect the efficiency of the other stages of the pipeline. For example, the number of calls to the event mechanism remains the same.

Table 3.11 also shows the results of the coarse-grained tuning for watpipe-b. The experiments show the best performance for watpipe-b is around 100 writer tasks and 20,000 connections (2.31). However, the performance of watpipe-b stabilizes around 100 writer tasks and 15,000 connections, so additional experiments were run based on this general vicinity and are presented in Table 3.12(b). For this table, the number of writer tasks were varied from 35 to 150 and the maximum number of connections from 13,000 to 17,000. This table also shows that the best performance occurs with 75 writer tasks and 15,000 connections (2.32), with a peak throughput of 1365 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1323 Mbps at 30,000 requests per second.

Similar to the other experiments with blocking sendfile, all experiments with fewer than 25 threads (writer tasks) did not verify. While experiments with 25 writer tasks did verify, all these experiments had tails that dropped by more than 20% as indicated in red in the table. Similarly, with 25,000 connections or more, 50 writer tasks are insufficient. All experiments with at least 50 writer tasks verified. Once the server had at least 75 writer tasks, adding further writers did not improve performance. As more writer tasks are added, performance did not drop appreciably, but eventually the performance would drop as the overhead of adding more writer tasks becomes larger.

The blocking version requires more threads due to blocking sendfile, however, the additional threads consume more resources and so the performance is lower than the non-blocking version. Since the system is under memory pressure, watpipe-nb has an advantage as it achieves its best performance with fewer writers. However, these memory differences do not entirely account for the difference in performance. For example, in Table 3.12 with 75 writers and 15,000 connections the performance of watpipe-nb is 2.35 while the performance of watpipe-b is 2.32. As these two configurations have similar memory footprints, memory is not the only difference. Examining the vmstat output at 15,000 requests per second reveals that the blocking server has 1% more user time and 2.2 times more context switching than the non-blocking server. The difference seems to be that a non-blocking writer thread can do more work before being context switched than a blocking writer thread, resulting in lower overhead for the non-blocking server. While these overheads are small, they do affect performance by 1–2% at peak and beyond.

For the watpipe-b experiments, the average I/O wait drops to zero once there are at least 75 writers. Similar to the versions of *μ*server using blocking sendfile, watpipe-b is also self limiting. For request rates of 10,000 requests per second or higher, with 100 processes and 25,000 or more simultaneous connections the number of requests read per second stabilizes.

3.10.4 Server Comparison

The previous sections examined the performance of different server architectures with multiple implementations of each under the same workload and environment by running experiments to tune each server in order to find its best performing configuration. In this section, the best configurations for each server are compared. As discussed earlier, the differences among the servers were minimized, and hence, the remaining differences are related to architecture and not other factors, such as caching strategies, event mechanisms, etc.

Figure 3.6 presents the best performing configuration for each server-architecture implementation: caching Knot (knot-c), non-blocking Knot (knot-nb), blocking Knot (knot-b), *μ*server non-

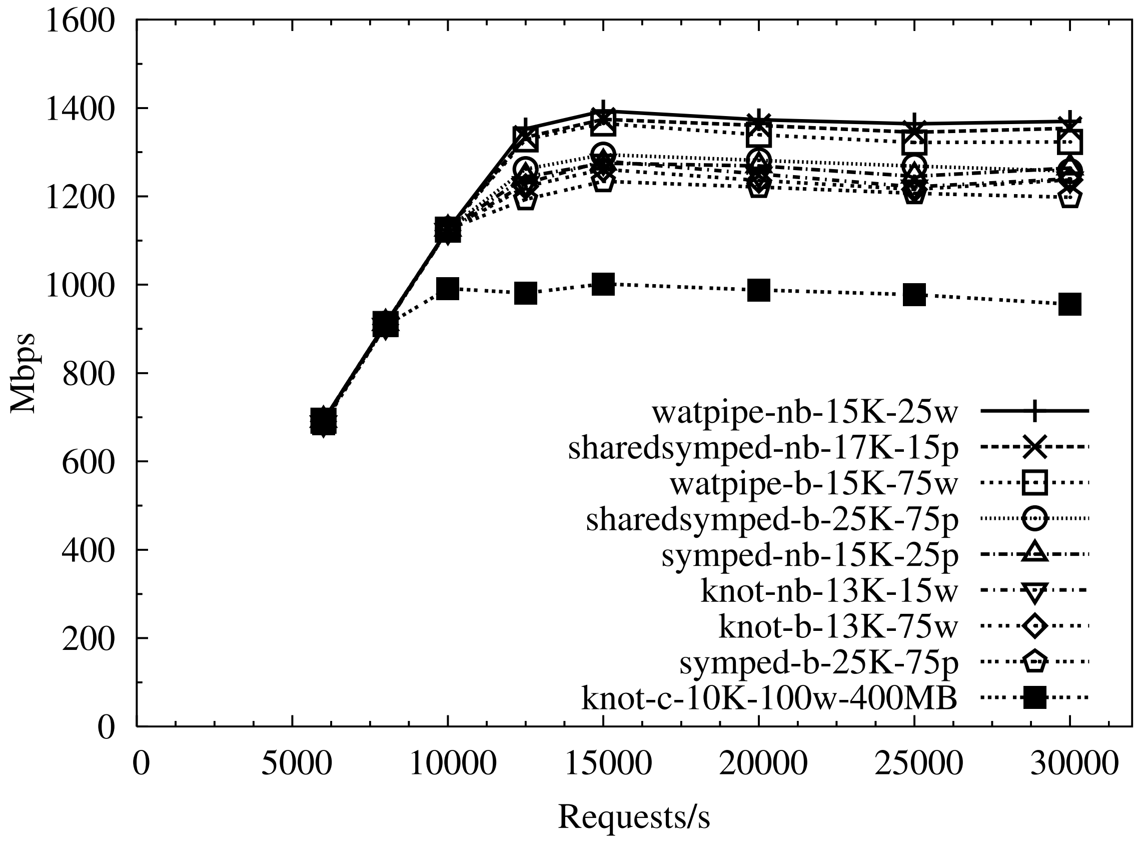


Figure 3.6: Throughput of different architectures - 1.4 GB

Server	Rank
watpipe-nb	1
sharedsymped-nb	2
watpipe-b	3
symped-nb	4
sharedsymped-b	4
knot-nb	5
knot-b	5
symped-b	6
knot-c	7

Table 3.13: Ranking of server performance - 1.4 GB

blocking SYMPED (symped-nb), *μ*server blocking SYMPED (symped-b), *μ*server non-blocking shared-SYMPED (sharedsymped-nb), *μ*server blocking shared-SYMPED (sharedsymped-b), non-blocking WatPipe (watpipe-nb) and blocking WatPipe (watpipe-b). The legend in Figure 3.6 is ordered from the best performing server at the top to the worst at the bottom. Excluding knot-c, peak server performance varies by about 11% (1234–1393 Mbps), indicating all the servers can do an excellent job.

Table 3.13 ranks the performance of the servers for the 1.4 GB workload. The ordering is determined by the area under the performance curve, with larger areas representing better performance. The best performing configuration for each server is run three times. Then Tukey's Honest Significant Difference test is used to determine, based on an analysis of variance, which mean areas are significantly different from one another with a 95% confidence level. The servers are then ranked based on mean area, with servers without a significant difference being grouped together.

The top performer is watpipe-nb, followed by sharedsymped-nb and watpipe-b; however, the difference in performance among the servers is small. The next grouping consists of sharedsymped-b and symped-nb. The sendfile version of the Knot servers, knot-nb and knot-b are next, followed closely by symped-b. Finally at the bottom is knot-c. Comparing the performance of the best version of WatPipe and the best version of *μ*server, watpipe-nb has a 1% higher peak at 15,000 requests per second and 1% higher performance after saturation at 30,000 requests per second, so the performance of these two servers is basically identical. Comparing the performance of the best version of WatPipe and the best versions of Knot, watpipe-nb has a 9% higher peak at 15,000 requests per second and 10% higher performance at 30,000 requests per second. The watpipe-nb and watpipe-b servers have performance within about 2–4% of each other. Sharedsymped-nb has a 6% higher peak at 15,000 requests per second and 8% higher performance at 30,000 requests per second than sharedsymped-b. Between the blocking *μ*server and non-blocking *μ*server versions is a larger gap; symped-nb has a 3% higher peak at 15,000 requests per second and 6% higher performance at 30,000 requests per second. The non-blocking and blocking versions of Knot have around the same performance. However, compared to knot-c, knot-nb has a 28% higher peak at 15,000 requests per second and 30% higher performance at 30,000 requests per second.

The performance of most of the servers is relatively close, but there are some interesting differences and similarities. In order to better understand the performance of the servers and to compare the servers, the best configuration of each server was profiled. Data is gathered by running OProfile and vmstat during an experiment where each server is subjected to a load of 15,000 requests per second. While the overhead of profiling does result in a performance penalty, this rate represents peak performance for most of the servers, and even with profiling, all the servers pass verification. OProfile periodically samples the execution of the program to determine where the system is spending time. As no unnecessary programs or services are running on the machine during an experiment, all profiling samples, including those in kernel and library code, can be legitimately attributed to the execution of the server. Additional statistics are gathered directly from the servers. The resulting data are summarized in Tables 3.14 and 3.15.

The tables are divided into four sections with the data for each server in a separate column. The first section lists the architecture of the server, the configuration parameters and the performance of the server in terms of both reply rate and throughput in megabits per second. In this section, "T/Conn" means thread per connection and "s-symped" means shared-SYMPED. The second section is a summary of the execution sampling data gathered by OProfile. The OProfile data consists of the percentage of samples that occurred in a particular function. From this data, it is possible to extrapolate how much CPU time the system is spending in each function. These functions are divided among 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 vmlinux and user-space sections are further divided into 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 light-weight 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 generates data about processes, memory, I/O, CPU activity, etc. For these experiments, vmstat was configured to sample the system every five seconds. The data presented in the table is an average of the sampled values gathered during the experiment. The row labelled "file-system cache" gives the average size of the Linux file-system cache in megabytes and 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 per second gathered directly from Capriccio. For each server, only the values where there is a significant difference among the servers are discussed.

The user-space total for all the Knot servers is 3% - 11% larger than the user-space total for the other servers. The difference is related to the fact that Knot requires a large number of threads ($\geq$ 10,000) to achieve good performance under this workload, resulting in additional overhead for user-level threading, I/O, synchronization and context switches. However, the size of this difference is misleading without taking into account some other values. First, the overhead for calling an event mechanism (epoll overhead) is lower with Knot than with the other servers. This difference is due to Knot's implementation of trying the call first and only using an event mechanism if the call cannot complete without blocking. In most cases, the call completes successfully the first time, resulting in rather sparse interest sets. Hence, using epoll instead of select or poll is more efficient [45]. Second, Capriccio calls into the kernel using syscalls and bypasses using libc for I/O. So the libc overhead from the other servers is combined into the thread overhead for the Knot servers. Hence, the extra overhead of running with a large number of threads is actually lower than it may appear.

As can be seen from Figure 3.6, the performance of knot-c is much lower than the other servers, even the other Knot servers. The OProfile data reveals that kernel data copying is quite large for knot-c at 17.46%, but only around 1% for the other servers. This overhead is virtually eliminated in knot-nb and knot-b by moving away from an application data cache and using sendfile instead, suggesting that maintaining an application cache is not a good technique even aside form the data duplication problems with maintaining two separate file caches. The next section, where the entire file set can fit into memory, examines if performance is comparable when the file-system cache can be bypassed or if operations that reduce data copying between the kernel and user-space are needed. Note that larger OProfile values are not always an indication of a problem or inefficiency. In some cases, the differences between knot-c and both knot-nb and knot-b, e.g., larger e1000 values, are related to differences in throughput. However, there are issues related to duplication of data between the application and file-system cache that make it difficult to confirm the hypothesis that high data copying overheads make write more expensive than sendfile. This observation is revisited in the next section.

Another difference among the servers is the average size of their file-system cache. As knot-c uses an application data-cache, its file-system cache is smaller than the other servers. For the servers using user-level threading, there is no significant difference in file-system cache despite the blocking version of the server requiring more kernel threads since the stacks are small and the address space is shared. Knot-nb and knot-b have virtually the same size file-system cache as do watpipe-nb and watpipe-b. For these servers, the non-blocking and blocking versions have approximately the same size file-system cache and almost the same throughput. For the μserver versions, the size of the difference depends on the amount of sharing among the processes. With SYMPED, the non-blocking version has a file-system cache of 1106 MB versus 939 MB for the blocking version, which is a reduction of about 167 MB for the 50 additional processes required by the blocking version. With shared-SYMPED, the non-blocking version has a file-system cache of 1203 MB versus 1105 MB for the blocking version. This is a reduction of about 98 MB for the 60 additional processes required by the blocking version. For both servers, the non-blocking version of the server has better performance. Even with the same number of processes, sharedsymped-b has a 166 MB larger file-system cache than symped-b, resulting in the shared-SYMPED servers having better performance than their SYMPED counterparts. Similarly, watpipe-b has an 82 MB larger file-system cache than sharedsymped-b, resulting in WatPipe having the best performance among the blocking servers.

Finally, it is interesting to note that the non-blocking servers with shared data, which require locking, also appear to have less context switching. While WatPipe does not have an alternative version to compare with, watpipe-nb has the second lowest amount of context switching at 933 context-switches per second. More interestingly, sharedsymped-nb has 845 context-switches per second on average compared to symped-nb with 3475 context switches per second. The scheduling overhead in the profiling tables show some of the direct overheads of context-switching; higher levels of kernel context-switching result in larger scheduling-overhead values. Both knot-nb and knot-b have around 20,000 kernel context-switches per second as well as around 20,000 user-level context switches per second. While knot-nb and knot-b have reasonable performance, their scheduling overheads are higher, indicating that high levels of context-switching do result in higher overheads. However, at these levels, context-switching does not appear to be a major problem. But, the differences in context switching among similar servers is important as it may still give insight into the behaviour of the servers. For example, with shared-SYMPED, fewer context switches could indicate that more data is being returned from epoll, resulting in processes executing longer between context switches.

Server Arch Write Sockets	**Knot-cache** **T/Conn** **non-block**	**Knot** **T/Conn** **non-block**	**Knot** **T/Conn** **block**	**userver** **symped** **non-block**	**userver** **symped** **block**
Max Conns Workers/Procs/Writers Other Config	**10K** **100w** **400MB**	**13K** **15w**	**13K** **75w**	**15K** **25p**	**25K** **75p**
Reply rate Tput (Mbps)	8002 952	9839 1177	9627 1149	9847 1174	9444 1127
OPROFILE DATA					
vmlinux total %	**65.34**	**57.45**	**59.00**	**63.25**	**64.09**
networking	22.46	27.95	28.22	28.74	28.81
memory-mgmt	7.16	6.93	6.66	7.38	7.53
file system	3.22	4.78	4.67	5.08	5.68
kernel+arch	5.59	5.98	6.30	8.52	8.00
epoll overhead	1.68	2.21	2.18	5.35	5.21
data copying	17.46	0.64	0.66	1.07	0.99
sched overhead	1.64	2.07	2.99	0.66	1.08
others	6.13	6.89	7.32	6.45	6.79
e1000 total %	**18.32**	**22.15**	**21.51**	**23.08**	**21.89**
user-space total %	**14.13**	**18.18**	**17.3**	**7.78**	**7.28**
thread overhead	6.24	10.09	9.2	0.00	0.00
event overhead	0.00	0.00	0.00	2.76	2.41
application	7.89	8.09	8.1	5.02	4.87
libc total %	**0.01**	**0.02**	**0.02**	**3.62**	**4.33**
other total %	**2.20**	**2.20**	**2.17**	**2.27**	**2.41**
VMSTAT DATA					
waiting %	0	0	0	0	0
file-system cache (MB)	795	1168	1168	1106	939
ctx-sw/sec (kernel)	2705	22,280	19,833	3475	4216
SERVER STATS					
ctx-sw/sec (user)	12,380	22,290	19,455		

Table 3.14: Server performance statistics gathered under a load of 15,000 requests per second - 1.4 GB

Server Arch Write Sockets	**userver** **s-symped** **non-block**	**userver** **s-symped** **block**	**WatPipe** **pipeline** **non-block**	**WatPipe** **pipeline** **block**
Max Conns Workers/Procs/Writers Other Config	**17K** **15p**	**25K** **75p**	**15K** **25w**	**15K** **75w**
Reply rate Tput (Mbps)	10,606 1267	10,028 1198	10,769 1286	10,490 1251
OPROFILE DATA				
vmlinux total %	**62.47**	**63.96**	**61.19**	**62.42**
networking	29.93	29.96	29.70	29.49
memory-mgmt	7.36	7.47	8.12	7.71
file system	4.41	4.31	4.36	4.32
kernel+arch	7.41	7.60	7.12	7.56
epoll overhead	5.80	5.59	4.75	4.58
data copying	1.10	1.05	0.83	0.84
sched overhead	0.28	1.18	0.56	1.25
others	6.18	6.80	5.75	6.67
e1000 total %	**24.23**	**22.42**	**24.06**	**22.51**
user-space total %	**7.91**	**7.63**	**10.84**	**11.21**
thread overhead	0.00	0.00	4.51	5.19
event overhead	2.83	2.63	2.44	2.17
application	5.08	5.00	3.89	3.85
libc total %	**3.31**	**3.88**	**1.86**	**1.78**
other total %	**2.08**	**2.11**	**2.05**	**2.08**
VMSTAT DATA				
waiting %	0	0	0	0
file-system cache (MB)	1203	1105	1187	1187
ctx-sw/sec (kernel)	845	4568	933	5078
SERVER STATS				
ctx-sw/sec (user)				

Table 3.15: Server performance statistics gathered under a load of 15,000 requests per second - 1.4 GB

3.11 4 GB

For these experiments, the system was configured with 4 GB of memory. In actual fact, the amount of available memory is around 3.7 GB due to parts of the address space being reserved for hardware devices. The idea behind these experiments is to examine performance when the entire file set is in memory, thus eliminating disk I/O. Given the size of the file set, 2.1 GB, 3.7 GB of memory is sufficient for these experiments. While the notion of having the file set in memory has different meanings for the various servers, ideally, all the servers should be able to achieve their maximum throughput under this configuration. As well, the expectation is that one or very few kernel threads are necessary for the servers using non-blocking I/O, since with this setup no overlapping of disk I/O and CPU execution is possible as the file set is cached and I/O waiting should be zero. However, in order to examine the tuning sensitivity of the servers, a large range of parameters are presented for each server even though the variation is small in some cases.

3.11.1 Tuning Knot

Tuning was again performed for knot-c, knot-nb and knot-b. For all servers, the parameters tuned are the number of threads and the number of worker tasks. Additionally, the size of the application cache is tuned for knot-c.

3.11.1.1 Knot-c

For the knot-c experiments, two cache sizes are used, 1000 MB and 2500 MB. As discussed earlier, the size of the file set is approximately 2.1 GB. With 1000 MB of application cache, the entire file set fits into the file-system cache but not the application cache. With 2500 MB of application cache, the entire file set no longer fits into the file-system cache but does fit into the application cache. As the point of these experiments is to eliminate disk I/O, another strategy is used when the application cache is set to 2500 MB. In this case, the file set is preloaded into the application cache using cache warming. To perform cache warming for knot-c, the file data for the entire file set is read into the application cache before the experiment begins. Hence, during the experiment all file requests can be serviced directly from the application cache without requiring any disk I/O. Note that using cache warming is an extra benefit for knot-c because the files do not need to be opened during the experiment as the file data is pre-cached on startup, resulting in reduced overhead. While it is possible to eliminate this difference by performing application cache-warming for the other server experiments, cache-warming is not done for the other server experiments because it would eliminate interesting differences among the servers with respect to shared versus non-shared application caches.

	Cache size in MB							
	10,000 threads		15,000 threads		20,000 threads		25,000 threads	
Workers	1000	2500	1000	2500	1000	2500	1000	2500
1	1.82	1.82	2.05	**2.10**	2.05	2.10	2.04	2.10
5	1.82	1.82	2.05	2.10	2.05	2.10	2.04	2.10
10	1.82	1.82	2.05	2.10	2.05	2.10	2.05	2.10
25	1.82	1.81	2.05	2.10	2.04	2.10	2.04	2.10
50	1.81	1.81	2.04	2.09	2.04	2.10	2.04	2.09
100	1.81	1.81	2.04	2.10	2.04	2.09	2.04	2.09

Table 3.16: Knot cache initial experiments - 4 GB

	2500 MB			
	Number of Threads			
Workers	13,000	15,000	18,000	20,000
1	2.10	2.10	2.10	2.10
5	**2.11**	2.11	2.10	2.10
10	2.10	2.11	2.10	2.10
25	2.10	2.10	2.10	2.10

Table 3.17: Knot cache fine tune experiments - 4 GB

Table 3.16 shows the results of the coarse-grained tuning for knot-c after verification. Each row represents a different number of workers from 1 to 100. The columns are separated into four sections with the results for 10,000 threads in section one, 15,000 threads in section two, 20,000 threads in section three and 25,000 threads in section four. In each section, the two columns correspond to the two different cache sizes being tested, i.e., 1000 or 2500 MB.

The experiments show the best performance is around 1 worker, at least 15,000 threads and 2500 MB of application cache (2.10). Based on this general vicinity, additional experiments were run and are presented in Table 3.17. In this table, the cache size is restricted to 2500 MB. Each row represents a different number of workers from 1 to 25 and each column represents a different number of threads from 13,000 to 20,000. This table shows that the best performance occurs with 13,000 threads, 5 workers and 2500 MB of application cache (2.11), with a peak throughput of 1203 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1192 Mbps at 30,000 requests per second. Note, since there is no disk I/O, the additional workers are doing very little work (some polling). Hence, the experiments are virtually identical, so the results for all the experiments are very close, and it is likely just experimental variation that resulted in this particular configuration giving the best performance for this run. This result represents an improvement of approximately 20% at peak and 25% for the sustained throughput over the best results obtained with knot-c for the 1.4 GB experiments.

However, even without disk I/O the performance of knot-c is still slightly worse than the performance of the other servers running with only 1.4 GB of memory; its performance is just below the level achieved by symped-b with 1.4 GB of memory. Eliminating cache duplication problems in these experiments should allow for a better comparison of using write with an application cache versus using zero-copy sendfile with the file-system cache. While there may be duplicated data between the application cache and the file-system cache, since the entire file-set is preloaded into the application cache, duplicated data in the file-system cache does not effect the performance of knot-c for these experiments.

Both Table 3.16 and Table 3.17 show that after at least 13,000 threads there is not much variation in the results. Only as the number of worker tasks gets larger than 25 is there a slight but consistent drop in throughput. As the memory overhead is irrelevant since there is enough memory to eliminate disk I/O and swapping, one obvious overhead is the amount of context switching. However, this overhead is small and does not affect performance greatly. For example, at 2500 MB of application cache and 15,000 threads under a load of 30,000 requests per second, the average number of kernel context switches per second for 1 worker is 34 and for 100 workers is 1024. It is likely the time spent polling the empty worker (disk I/O) queue results in more overhead than the context switching.

3.11.1.2 Knot-nb and knot-b

Table 3.18(a) shows the results of the coarse-grained tuning for knot-nb. Each row in the table represents a different number of workers from 1 to 100 and each column represents a different number of threads from 10,000 to 25,000. For these experiments, the best performance occurs at 15,000 threads and at least 1 worker (2.26). Based on this general vicinity, additional experiments were run and are presented in Table 3.19(a). For this table, the number of workers are varied from 1 to 10 and the number of threads from 13,000 to 17,000. For knot-nb, the best performance occurs with 15,000 threads and 1 worker (2.29), with a peak throughput of 1339 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1320 Mbps at 30,000 requests per second. This result represents an improvement of approximately 5% at peak and 6% for the sustained throughput over the best results obtained for the 1.4 GB experiments. This improvement is much smaller than those obtained by knot-c; however, knot-c benefited from more than just eliminating disk I/O. Not only did the entire file set fit into the application cache, reducing tension with the file-system cache, but knot-c had the added benefit of application cache warming.

It is interesting to note that 10,000 connections is again an insufficient number of threads and lim-

Workers	Maximum Number of Threads			
	10,000	15,000	20,000	25,000
1	1.82	**2.26**	2.24	×
5	1.82	2.26	2.24	2.24
10	1.83	2.26	2.24	2.23
25	1.83	2.26	2.24	2.24
50	1.83	2.25	2.24	2.24
100	1.83	2.25	2.24	2.24

(a) non-blocking sendfile

Workers	Maximum Number of Threads			
	10,000	15,000	20,000	25,000
1	×	×	×	×
5	×	×	×	×
10	0.99	×	×	×
25	1.70	1.75	1.50	1.38
50	1.82	2.20	2.14	2.09
100	1.83	2.22	**2.23**	2.23
150	1.83	2.22	2.22	2.23

(b) blocking sendfile

Table 3.18: Knot sendfile initial experiments - 4 GB

Workers	Maximum Number of Threads		
	13,000	15,000	17,000
1	2.26	**2.29**	2.27
3	2.26	2.28	2.27
5	2.26	2.28	2.27
10	2.26	2.28	2.27

(a) non-blocking sendfile

Workers	Maximum Number of Threads			
	20,000	25,000	30,000	35,000
100	**2.26**	2.25	2.25	2.25
150	2.26	2.25	2.25	2.25
200	2.25	2.25	2.25	2.25

(b) blocking sendfile

Table 3.19: Knot sendfile fine tune experiments - 4 GB

its performance. When the number of simultaneous connections is capped at 10,000, throughput never exceeds 1036 Mbps, and the condensed area is approximately 1.84 for these experiments. This value is almost identical to the peak throughput observed with 1.4 GB of memory, suggesting performance is limited by only having 10,000 threads and not other overheads such as memory or disk I/O.

Finally, the tuning stability seen in the knot-c experiments when adding workers continues with the knot-nb experiments. However, increasing the number of threads beyond 15,000 results in a gradual decline in performance.

When knot-c is run with 2500 MB of cache, problems related to the size of the application cache versus the file-system cache are eliminated. In this case, the entire file set fits into the application cache. Hence, any differences between knot-c and knot-nb are likely related to the difference in efficiency between write versus zero-copy sendfile. It appears that sendfile is more efficient and results in better performance even with the benefit that knot-c gets from cache warming. This observation is explored further in the Server

Comparison section (3.11.4).

Table 3.18(b) shows the results of the coarse-grained tuning for knot-b after verification. Each row in the table represents a different number of workers from 1 to 150 and each column represents a different number of threads from 10,000 to 25,000. For these experiments, the best performance occurs at around 20,000 threads and 100 workers (2.23). Based on this general vicinity, additional experiments were run and are presented in Table 3.19(b). For this table, the number of workers are varied from 100 to 200 and the number of threads from 20,000 to 35,000. For knot-b, the performance of the experiments in Table 3.19(b) are similar enough that the least resource intensive configuration is chosen as the best. The best performance occurs with 20,000 threads and 100 workers (2.26), with a peak throughput of 1310 Mbps occurring at 30,000 requests per second. In this case, the peak occurs at the highest rate run but the throughput at 15,000 requests per second is 1308 Mbps at which point performance is relatively level. This result represents an improvement of approximately 4% at 15,000 requests per second and 6% for the sustained throughput over the best results obtained for the 1.4 GB experiments. This improvement is in line with the improvements for knot-nb and smaller than those seen by knot-c.

Unlike knot-nb, even though the entire file set fits into the file-system cache, more workers are required compared to knot-b running with 1.4 GB of memory. In this case, however, the workers are only blocked waiting to send the contents of larger files and not due to disk I/O. The larger number of workers are a function of the higher throughput and the increased number of simultaneous connections. It is also interesting to note that similar to the knot-b experiments with 1.4 GB of memory, more than 25 workers are needed, otherwise experiments do not verify or performance drops of more than 20% occur after peak.

While there is no memory pressure, knot-b does not perform as well as knot-nb. The only significant difference is the larger number of workers required by knot-b. Based on vmstat output, one big difference between the servers is kernel context-switching overhead. For their corresponding best performing configurations, at a request rate of 30,000 requests per second, knot-nb has an average of 39 context switches per second and knot-b has 6875 context switches per second. While the context switching overhead is not very large in either case, the amount of time spent polling the worker (disk I/O) queue is much higher with knot-b. While knot-b requires a larger number of workers to handle cases where many large files are being sent concurrently, at other times these extra workers simply consume CPU time polling the worker queue because they spin.

As well, knot-b appears to be self-limiting. Based on server statistics, the number of simultaneous connections does not appear to exceed 30,000 for any of the experiments in Table 3.19(b). Hence, further increases to the number of threads is not helpful.

3.11.2 Tuning *μ*server

Experiments were run to tune the four versions of *μ*server: symped-nb, symped-b, sharedsymped-nb and sharedsymped-b. For all versions of *μ*server, the parameters tuned are the maximum number of simultaneous connections and the number of processes.

3.11.2.1 Symped-nb and symped-b

Table 3.20(a) shows the results of the coarse-grained tuning for symped-nb after verification. Each row in the table represents a different number of processes from 1 to 100 and each column a different maximum number of connections from 10,000 to 25,000. The experiments show the best performance for symped-nb is around 1 process and 15,000 maximum number of connections (2.44). Based on this general vicinity, additional symped-nb experiments were run and are presented in Table 3.21(a). For this table, the number of processes are varied from 1 to 4 and the maximum number of connections from 13,000 to 20,000. Table 3.21(a) shows that the best performance for symped-nb occurs with 1 process and 18,000 connections (2.45), with a peak throughput of 1444 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1437 Mbps at 30,000 requests per second. This result represents an improvement of approximately 13% at 15,000 requests per second and 14% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

Moving from an actual SYMPED server to a SPED server represents a significant reduction in overhead for symped-nb. Efficiencies are gained by only having a single process, such as fewer and hence more efficient calls to the event mechanism, very few context switches and the *μ*server SPED servers are self-limiting so verification problems are less likely as the maximum number of connections increase. These types of efficiencies combined with other advantages of only having a single thread, such as, little to no contention for resources, reduced scheduling overheads and better usage of hardware caches, result in a performance advantage for the SPED servers.

The SPED runs are self-limiting; the maximum number of simultaneous connections never exceed 20,000. As well, the best performance occurs with 1 process as there is no blocking for disk I/O. Aside from a performance boost when the maximum number of connections is at least 15,000, the results for 4 GB are very similar to results for 1.4 GB with respect to verification. As discussed earlier, once the I/O wait is zero, adding additional processes does not improve performance.

Symped-nb experiments with 15,000 maximum connections and between 3 and 10 processes did not verify, but other experiments with 15,000 maximum connections did verify. These verification problems occur at 15,000 requests per second because the server is close to the edge of its tuning range for the maximum number of connections. As discussed earlier, verification failures occur because the server

Procs	Maximum Number of Connections			
	10,000	15,000	20,000	25,000
1	1.83	**2.44**	2.43	2.43
5	1.84	×	×	×
10	1.83	×	×	×
25	1.83	2.33	×	×
50	1.83	2.29	×	×
100	1.78	2.18	×	×

(a) non-blocking sendfile

Procs	Maximum Number of Connections				
	10,000	15,000	20,000	25,000	30,000
1	×	×	×	×	×
5	×	×	×	×	×
10	1.02	×	×	×	×
25	1.59	1.75	1.59	1.44	×
50	1.75	2.10	2.14	2.10	2.01
100	1.78	2.08	2.14	2.17	**2.18**

(b) blocking sendfile

Table 3.20: μserver SYMPED initial experiments - 4 GB

Procs	Maximum Number of Connections			
	13,000	15,000	18,000	20,000
1	2.28	2.44	**2.45**	2.44
2	2.28	2.32	×	×
3	2.28	×	×	×
4	2.28	×	×	×

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	25,000	30,000	35,000	40,000
100	2.19	**2.21**	2.21	2.21
125	2.17	2.18	2.19	2.19
150	2.15	2.16	2.17	2.17

(b) blocking sendfile

Table 3.21: μserver SYMPED fine tune experiments - 4 GB

reads more requests than it can process, especially for higher request rates, so timeouts occur with large files. With one process, event polling occurs at specific times and the requests available at that time are processed before the event mechanism is called again. When there are several processes, event polling occurs independently for each process over a larger span of time, resulting in more requests available for reading. As the overall number of requests read becomes larger than the server can handle, verification problems occur. However, as the number of processes increases, the server operates less efficiently and eventually more requests time out before being read. Since throughput decreases as the number of processes increases, the number of requests handled by the server must also decrease. Because there is a direct correlation between the number of requests processed and the throughput of the server, if the number of requests handled by the server did not decrease, then additional read requests must time out, since the throughput is lower, resulting in verification failures. Therefore, while the experiments with more than 10 processes verify, performance decreases.

Table 3.20(b) shows the results of the coarse-grained tuning for symped-b. Each row in the table represents a different number of processes from 1 to 100 and each column a different maximum number of connections from 10,000 to 30,000. The experiments show the best performance for symped-b is around 100 processes and 30,000 maximum number of connections (2.18). Based on this general vicinity, additional symped-b experiments were run and are presented in Table 3.21(b). For this table, the number of processes are varied from 100 to 150 and the maximum number of connections from 25,000 to 40,000. Table 3.21(b) shows that the best performance for symped-b occurs with 100 processes and 30,000 connections (2.21), with a peak throughput of 1284 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1263 Mbps at 30,000 requests per second. This result represents an improvement of approximately 4% at 15,000 requests per second and 5% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

Similar to knot-b, more than 25 processes are needed, otherwise experiments do not verify or drops in throughput larger than 20% occur after peak. The requirement for more than 1 process means that symped-b does not perform as efficiently as symped-nb running SPED. As discussed earlier, with no memory pressure a SPED server is more efficient than running with multiple processes. For example, at a rate of 30,000 requests per second the best symped-b configuration made 128,365 calls to the event mechanism compared to 103,696 calls by symped-nb, a difference of approximately 24%. (See Section 3.11.4 for profiling data related to the efficiencies of running with a single process.) As well, since SYMPED processes are symmetric, each process performs all the functions of a complete server, so additional processes tend to have a larger negative effect on performance than asymmetric tasks like the worker threads in Knot. Therefore, the performance difference between symped-nb and symped-b is larger than the performance difference between knot-nb and knot-b. For the same reason, the SYMPED experiments for this workload exhibit less tuning stability as the number of processes are increased compared to the Knot experiments when the workers are increased.

3.11.2.2 Sharedsymped-nb and sharedsymped-b

Table 3.22(a) shows the results of the coarse-grained tuning for sharedsymped-nb after verification. Each row in the table represents a different number of processes from 1 to 100 and each column a different maximum number of connections from 10,000 to 25,000. The experiments show the best performance for sharedsymped-nb is around 1 process and 15,000 maximum number of connections (2.46). Based on this general vicinity, additional sharedsymped-nb experiments were run and are presented in Table 3.23(a). For this table, the number of processes are varied from 1 to 4 and the maximum number of connections from 15,000 to 30,000. Table 3.23(a) shows that the best performance for sharedsymped-nb occurs with 1 process and 20,000 connections (2.46), with a peak throughput of 1457 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1442 Mbps at 30,000 requests per second.

This result represents an improvement of approximately 6% at 15,000 requests per second and 6% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

Similar to symped-nb, the best performance for sharedsymped-nb occurs when it is running SPED. As expected, the performance of symped-nb and sharedsymped-nb running SPED is approximately the same and within the range of experimental variation. Hence, the improvement for sharedsymped-nb is less than the improvement for symped-nb because sharedsymped-nb has better performance with 1.4 GB of memory. With 1.4 GB of memory, the sharedsymped-nb server benefits from less runtime overhead due to fewer processes and a smaller memory footprint due to a shared cache-table and fewer processes, compared to symped-nb. In moving to 4 GB of memory, performance gains result from needing only one process and no memory pressure in the system. These two changes are both smaller for sharedsymped-nb, resulting in a smaller performance improvement for sharedsymped-nb versus symped-nb.

The pattern of verification failures with 15,000 maximum connections is similar for both symped-nb and sharedsymped-nb, and occur for the same reason. Interestingly, the sharedsymped-nb experiment with 15,000 maximum connections and 10 processes in Table 3.22(a) does verify, however, it has lower throughput than the experiment with 25 processes. This unusual dip in performance occurs because the experiment with 10 processes is close to failing due to timeouts on large files. These timeouts result in a lower condensed area because only completed transfers are included in the calculation of throughput.

Table 3.22(b) shows the results of the coarse-grained tuning for sharedsymped-b after verification. Each row in the table represents a different number of processes from 1 to 100 and each column a different maximum number of connections from 10,000 to 30,000. The experiments show the best performance for sharedsymped-b is around 100 processes and 30,000 maximum number of connections (2.30). Based on this general vicinity, additional sharedsymped-b experiments were run and are presented in Table 3.23(b). For this table, the number of processes are varied from 75 to 150 and the maximum number of connections from 25,000 to 40,000. Table 3.23(b) shows that the best performance for sharedsymped-b occurs with 75 processes and 25,000 connections (2.31), with a peak throughput of 1356 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1311 Mbps at 30,000 requests per second. This result represents an improvement of approximately 5% at 15,000 requests per second and 4% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

Similar to symped-b, these improvements are not as large as those achieved by its non-blocking counterpart due to the efficiencies gained when sharedsymped-nb is run as a SPED server. It is interesting to note that there is approximately 2% idle time at 30,000 requests per second with 75 processes and 25,000 maximum connections. The amount of idle time drops to zero as the number of processes is increased but performance does not improve.

Similar to knot-b and symped-b, more than 25 workers are needed, otherwise experiments do not

Procs	Maximum Number of Connections			
	10,000	15,000	20,000	25,000
1	1.83	**2.46**	2.46	2.45
5	1.84	✗	✗	✗
10	1.83	2.35	✗	✗
25	1.84	2.41	✗	✗
50	1.83	2.36	✗	✗
100	1.81	2.26	✗	✗

(a) non-blocking sendfile

Procs	Maximum Number of Connections				
	10,000	15,000	20,000	25,000	30,000
1	✗	✗	✗	✗	✗
5	✗	✗	✗	✗	✗
10	1.01	✗	✗	✗	✗
25	1.58	1.75	1.57	✗	✗
50	1.75	2.17	2.21	2.16	2.05
100	1.80	2.18	2.27	2.29	**2.30**

(b) blocking sendfile

Table 3.22: μserver shared-SYMPED initial experiments - 4 GB

Procs	Maximum Number of Connections			
	15,000	20,000	25,000	30,000
1	2.45	**2.46**	2.45	2.46
2	2.33	✗	✗	✗
3	✗	✗	✗	✗
4	✗	✗	✗	✗

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	25,000	30,000	35,000	40,000
75	**2.31**	2.30	2.27	2.21
100	2.29	2.30	2.30	2.30
125	2.28	2.28	2.28	2.29
150	2.25	2.27	2.28	2.29

(b) blocking sendfile

Table 3.23: μserver shared-SYMPED fine tune experiments - 4 GB

verify or drops in throughput larger than 20% occur after peak. As expected, the larger number of workers means sharedsymped-b cannot perform as efficiently as sharedsymped-nb running SPED. By comparison, the best sharedsymped-b configuration at a rate of 30,000 requests per second makes 206,216 calls to the event mechanism compared 104,941 calls to the event mechanism for the best sharedsymped-nb running SPED. As well, the symmetric processes with shared-SYMPED result in a larger performance difference between sharedsymped-nb and sharedsymped-b versus knot-nb and knot-b.

A reasonable expectation is that sharedsymped-b would perform worse than symped-b for this workload. Since the system is not under memory pressure, a shared cache offers no advantages and the additional overhead of locking should adversely affect performance. However, the performance of sharedsymped-b is better than the performance of symped-b. It has approximately 6% better performance at peak and 4% for sustained throughput. This performance difference is examined further in Section 3.11.4 with the help of profiling data.

3.11.3 Tuning WatPipe

Experiments were run to tune the two versions of WatPipe: watpipe-nb and watpipe-b. For both versions of WatPipe, the parameters tuned are the maximum number of simultaneous connections and the number of writer tasks.

Table 3.24(a) shows the results of the coarse-grained tuning for watpipe-nb. Each row in the table represents a different number of writers from 1 to 100 and each column a different maximum number of connections from 10,000 to 25,000. The experiments show the best performance for watpipe-nb is around 1 writer and 15,000 maximum number of connections (2.43). Based on this general vicinity, additional watpipe-nb experiments were run and are presented in Table 3.25(a). For this table, the number of writers are varied from 1 to 4 and the maximum number of connections from 15,000 to 23,000. Table 3.25(a) shows that the best performance for watpipe-nb occurs with 1 writer and 18,000 connections (2.45), with a peak throughput of 1452 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1439 Mbps at 30,000 requests per second. This result represents an improvement of approximately 4% at 15,000 requests per second and 5% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

The overall performance of watpipe-nb is similar to sharedsymped-nb. The expectation is that, in the absence of disk I/O, a SPED server is more efficient than a server with multiple threads. Hence, the fact that watpipe-nb manages to achieve similar performance to both of the SPED servers is surprising. However, WatPipe also has certain efficiencies gained by only having multiple writer tasks. What is not surprising is that, unlike the two non-blocking versions of μserver, watpipe-nb has stable performance with up to 100 writer tasks even though only one writer task is required because the cost of additional writers is negligible.

Table 3.24(b) shows the results of the coarse-grained tuning for watpipe-b. Each row in the table represents a different number of writers from 1 to 100 and each column a different maximum number of connections from 10,000 to 25,000. The experiments show the best performance for watpipe-b is around 100 writers and 15,000 maximum number of connections (2.33). Based on this general vicinity, additional watpipe-b experiments were run and are presented in Table 3.25(b). For this table, the number of writers are varied from 75 to 150 and the maximum number of connections from 13,000 to 20,000. Table 3.25(b) shows that the best performance for watpipe-b occurs with 75 writers and 15,000 connections (2.36), with a peak throughput of 1399 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 1357 Mbps at 30,000 requests per second. This result represents an improvement of approximately 2% at 15,000 requests per second and 3% for the sustained throughput over the best results obtained for the 1.4 GB experiments. Interestingly, the throughput of watpipe-b for the 1.4 GB workload, despite the presence of disk I/O, is close to its performance for the in-memory workload.

Writers	Maximum Number of Connections			
	10,000	15,000	20,000	25,000
1	1.77	**2.43**	2.41	✗
5	1.78	2.42	2.42	✗
10	1.78	2.42	2.41	✗
25	1.78	2.42	2.42	✗
50	1.78	2.41	2.41	✗
100	1.77	2.40	2.40	✗

(a) non-blocking sendfile

Writers	Maximum Number of Connections			
	10,000	15,000	20,000	25,000
1	✗	✗	✗	✗
5	✗	✗	✗	✗
10	1.01	0.98	✗	✗
25	1.68	1.83	1.53	1.40
50	1.78	2.32	2.23	2.16
100	1.77	**2.33**	2.32	2.32

(b) blocking sendfile

Table 3.24: WatPipe initial experiments - 4 GB

Writers	Maximum Number of Connections			
	15,000	18,000	20,000	23,000
1	2.44	**2.45**	2.44	✗
2	2.45	2.45	2.44	✗
3	2.44	2.45	2.45	✗
4	2.44	2.45	2.44	✗

(a) non-blocking sendfile

Writers	Maximum Number of Connections			
	13,000	15,000	18,000	20,000
75	2.24	**2.36**	2.36	2.35
100	2.23	2.36	2.36	2.35
125	2.23	2.36	2.36	2.35
150	2.23	2.35	2.36	2.35

(b) blocking sendfile

Table 3.25: WatPipe fine tune experiments - 4 GB

Despite a smaller increase in performance compared to symped-b and sharedsymped-b, watpipe-b has better performance than those servers. In the 1.4 GB experiments, watpipe-b had the advantage of a smaller memory footprint so under memory pressure it is reasonable that watpipe-b has better performance. However, for this set of experiments there is no memory pressure but watpipe-b still has better performance than symped-b and sharedsymped-b despite the fact that their non-blocking counterparts have similar performance. One disadvantage of the SYMPED approach is that all the server processes perform all the steps of a single SPED server. While this approach scales well for a small number of server processes, as the number of server processes gets larger, inefficiencies related to this architecture emerge. One such inefficiency is the number of calls to the event mechanism. For example, for the best blocking server configurations at 30,000 requests per second watpipe-b has 45,475 calls to the event mechanism, sharedsymped-b has 206,216 calls across 75 processes and symped-b has 128,365 calls across 100 processes. As mentioned earlier, the additional performance difference for symped-b is examined further in Section 3.11.4 with the help of profiling data.

3.11.4 Server Comparison

Figure 3.7 presents the best performing configuration for each server architecture implementation: caching Knot (knot-c), non-blocking Knot (knot-nb), blocking Knot (knot-b), μserver non-blocking SYMPED (symped-nb), μserver blocking SYMPED (symped-b), μserver non-blocking shared-SYMPED (sharedsymped-nb), μserver blocking shared-SYMPED (sharedsymped-b), non-blocking WatPipe (watpipe-nb) and blocking WatPipe (watpipe-b). The legend in Figure 3.7 is ordered from the best performing server at the top to the worst at the bottom. Excluding knot-c, peak server performance varies by about 13% (1284–1457 Mbps), indicating all the servers can do an excellent job.

Table 3.26 ranks the performance of the servers for the 4 GB workload. Again, based on a total of three runs for each server, Tukey's Honest Significant Difference test is used to differentiate the servers with a 95% confidence level. The servers are then ranked based on mean area.

The top performing servers are sharedsymped-nb, watpipe-nb and symped-nb, all with approximately the same performance. The next server is watpipe-b, which has slightly better performance than sharedsymped-b and knot-nb. The last two blocking servers, knot-b and symped-b, occur next with knot-c at the bottom again. The performance gap between the non-blocking and blocking servers is larger than with the 1.4 GB experiments. The best non-blocking server, sharedsymped-nb, compared to the best blocking server, watpipe-b, has a 4% higher peak at 15,000 requests per second and 6% higher performance after saturation at 30,000 requests per second. Compared to the 1.4 GB experiments, the throughput of the non-blocking servers improved more than the throughput of their blocking counterparts. This difference is a result of the number of threads required for the non-blocking servers decreasing while the number of threads required for the blocking servers increasing. Comparing sharedsymped-nb to the best Knot server, sharedsymped-nb has a 9% higher peak at 15,000 requests per second and 9% higher performance at 30,000 requests per second. This performance difference is approximately the same compared to the 1.4 GB experiments. The non-blocking version of knot compared to knot-c has an 11% higher peak at 15,000 requests per second and 11% higher performance at 30,000 requests per second. This difference is much lower than the difference at 1.4 GB and is a result of the additional benefits experienced by knot-c beyond eliminating disk I/O.

To better understand the performance of the servers, the best configuration of each server is profiled. The OProfile and vmstat data for these experiments are summarized in Tables 3.27 and 3.28. Perhaps the most interesting observation is that the OProfile data gathered for each server is very similar to the data gathered for the 1.4 GB experiments. Differences are small and generally related to throughput, higher networking, e1000 and epoll overhead values with 4 GB, and number of threads, lower scheduling overheads for the non-blocking servers with 4 GB. The biggest differences occur in the file-system cache-size and context switches per second values reported by vmstat. For the 4 GB experiments, the file-system

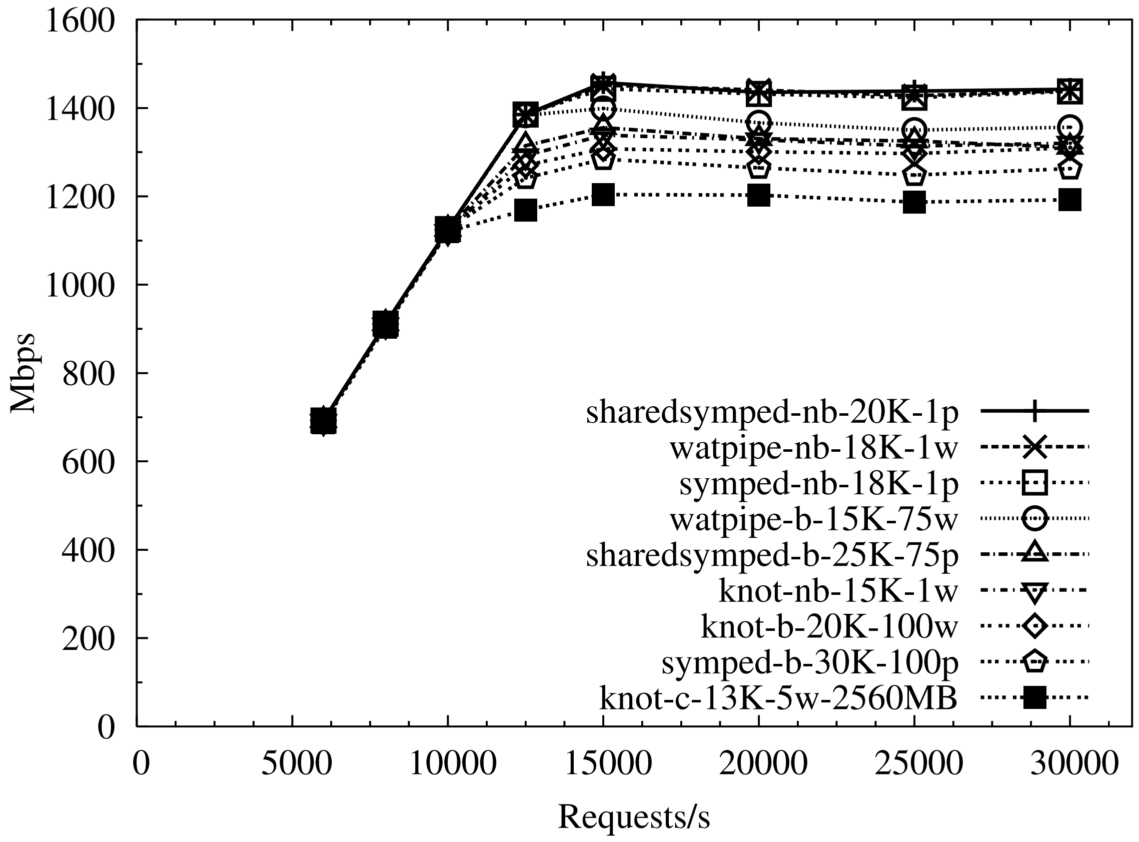


Figure 3.7: Throughput of different architectures - 4 GB

Server	Rank
symped-nb	1
sharedsymped-nb	1
watpipe-nb	1
watpipe-b	2
sharedsymped-b	3
knot-nb	4
knot-b	5
symped-b	6
knot-c	7

Table 3.26: Ranking of server performance - 4 GB

cache size is always large enough to contain the entire file set and any variations between the 4 GB experiments are not important. The exception is knot-c, which uses cache warming and an application cache large enough to contain the entire file set. The amount of context switching is directly related to the number of kernel threads and hence is generally lower for the non-blocking, 4 GB experiments. The exception is watpipe-nb, which has significantly more context switching than the SPED servers due to its pipeline design with dedicated threads for each stage. Despite having more than one thread, watpipe-nb performs as well as the SPED servers.

Several observations from the 1.4 GB experiments are true for these experiments as well. Knot-c again has high data-copying overheads (16.89%) due to using write, inhibiting the performance of the server. In general, Knot has higher user-space plus libc overheads than the other servers because a large number of user-level threads incurs high thread overheads. However, similar to the analysis for the 1.4 GB experiments, part of this additional overhead is mitigated by lower epoll overheads. Despite these overheads pushing the performance of Knot lower than the other servers, knot-nb and knot-b still manage to outperform symped-b.

While it is expected that the blocking version of a server has worse performance than its non-blocking counterpart, the performance of symped-b is especially poor. The idea behind shared-SYMPED is to reduce the memory footprint of the server to reduce the amount of disk I/O when there is memory pressure in the system. When there is no memory pressure, the expectation is that the overheads related to locking and contention on the shared application cache negatively affects performance. Hence, it is reasonable to expect that symped-b would have better performance than sharedsymped-b in these experiments but the actual performance of symped-b is worse. Examining the OProfile data and the server statistics at a rate of 15,000 request per second reveals some interesting observations. The time spent in the file-system category is about 36% larger for symped-b than for sharedsymped-b. As well, the server output shows the application cache hit rate for symped-b is 78% versus 99.5% for sharedsymped-b. For both servers, the application cache only contains file descriptors and HTTP headers, and is large enough to cache the entire file-set. The first time a file is requested, the file is not in the application cache (cache miss) so the server opens the file and stores the file descriptor and an associated HTTP header in its application cache. If a subsequent request occurs for the same file, the file is already in the application cache (cache hit) so no additional work is required to cache the file. The cache hit rate is calculated by dividing the total number of cache hits across every file by the total number of requests. With μserver SYMPED, each server process has a separate application cache so each process must individually open and create a cache entry for each file. With μserver shared-SYMPED, the cache is shared so each file is only opened and cached once for the entire server the first time any server process receives a request for the file. Because each symped-b process has its own separate application cache, each file needs to be looked up multiple times and it takes longer for each individual cache to become fully populated than for a single shared cache. Hence, there is significantly more file system activity and increased kernel locking and contention on the file system due to the larger number of I/O accesses with symped-b, resulting in poorer performance. For the experiment at 15,000 requests per second, symped-b has 807,719 cache misses versus 21,297 for sharedsymped-b, a difference of 786,422. An experiment comparing sharedsymped-b and symped-b with 100 processes and 30,000 connections with a warmed application cache, i.e., all the files in the file-set are opened before the experiment begins, thereby eliminating most of the difference in file system activity between the servers during the experiment, shows approximately equivalent performance for both servers.

The expectation is that eliminating disk I/O would result in the servers requiring fewer kernel threads to achieve their best performance. While this expectation holds true for the non-blocking servers, it does not hold true for the blocking servers. Instead, the higher throughput of the blocking servers means that more threads/processes are required to handle more simultaneous requests for large files. In fact, the number of threads required is dictated by the size of the socket buffers, the number of simultaneous requests for large files and the throughput of the server.

The additional processes required by the blocking servers are problematic as they reduce the efficiency of the server and increase the system overheads. For the SYMPED and shared-SYMPED servers, the OProfile data shows that the kernel overheads, especially related to scheduling, are larger for the blocking version of the servers. These increases are consistent across the servers and indicate that additional kernel threads in the system cause an increase in overheads.

Server Arch Write Sockets	**Knot-cache** **T/Conn** **non-block**	**Knot** **T/Conn** **non-block**	**Knot** **T/Conn** **block**	**userver** **symped** **non-block**	**userver** **symped** **block**
Max Conns	**13K**	**15K**	**20K**	**18K**	**30K**
Workers/Procs/Writers	**5w**	**1w**	**100w**	**1p**	**100p**
Other Config	**2560MB**				
Reply rate	9296	10,340	10,061	10,893	9672
Tput (Mbps)	1108	1233	1202	1301	1152
OPROFILE DATA					
vmlinux total %	**63.65**	**56.84**	**58.79**	**62.07**	**64.11**
networking	24.94	29.33	29.31	30.07	29.25
memory-mgmt	5.32	6.57	6.37	6.90	6.85
file system	2.55	4.81	4.58	4.24	5.74
kernel+arch	5.61	6.19	6.48	6.85	8.29
epoll overhead	1.96	2.34	2.21	7.11	5.37
data copying	16.89	0.63	0.71	1.16	1.00
sched overhead	1.13	0.96	2.40	0.06	1.13
others	5.25	6.01	6.73	5.68	6.48
e1000 total %	**20.41**	**23.15**	**22.19**	**24.87**	**21.92**
user-space total %	**14.29**	**18.27**	**17.15**	**8.14**	**7.36**
thread overhead	6.6	10.53	9.39	0.00	0.00
event overhead	0.00	0.00	0.00	2.95	2.35
application	7.69	7.74	7.76	5.19	5.01
libc total %	**0.01**	**0.01**	**0.02**	**3.19**	**4.57**
other total %	**1.64**	**1.73**	**1.85**	**1.73**	**2.04**
VMSTAT DATA					
waiting %	0	0	0	0	0
file-system cache (MB)	1309	2287	2291	2270	2272
ctx-sw/sec (kernel)	96	190	5910	68	4328
SERVER STATS					
ctx-sw/sec (user)	12,269	23,781	20,575		

Table 3.27: Server performance statistics gathered under a load of 15,000 requests per second - 4 GB

Server Arch Write Sockets	**userver** **s-symped** **non-block**	**userver** **s-symped** **block**	**WatPipe** **pipeline** **non-block**	**WatPipe** **pipeline** **block**
Max Conns Workers/Procs/Writers Other Config	**20K** **1p**	**25K** **75p**	**18K** **1w**	**15K** **75w**
Reply rate Tput (Mbps)	11,190 1334	10,347 1235	10,926 1305	10,621 1266
OPROFILE DATA				
vmlinux total %	**61.78**	**63.07**	**59.97**	**61.68**
networking	30.12	29.90	30.18	29.47
memory-mgmt	6.93	6.99	7.57	7.17
file system	4.18	4.23	4.24	4.29
kernel+arch	6.82	7.63	6.98	7.74
epoll overhead	6.83	5.55	4.79	4.44
data copying	1.14	1.09	0.81	0.87
sched overhead	0.06	1.16	0.29	1.32
others	5.70	6.52	5.11	6.38
e1000 total %	**25.12**	**23.18**	**25.23**	**22.89**
user-space total %	**8.13**	**7.85**	**11.13**	**11.78**
thread overhead	0.00	0.00	4.16	5.33
event overhead	2.92	2.65	2.62	2.33
application	5.21	5.20	4.35	4.12
libc total %	**3.21**	**3.99**	**2.03**	**1.84**
other total %	**1.76**	**1.91**	**1.64**	**1.81**
VMSTAT DATA				
waiting %	0	0	0	0
file-system cache (MB)	2295	2295	2291	2291
ctx-sw/sec (kernel)	68	4519	1058	6058
SERVER STATS				
ctx-sw/sec (user)				

Table 3.28: Server performance statistics gathered under a load of 15,000 requests per second - 4 GB

3.12 .75 GB

The final workload examined is on a system configured with .75 GB of memory. These experiments examine the performance of the servers when there is heavy disk I/O, so a server cannot completely eliminate I/O wait. The expectation is that these experiments should have significantly lower throughput because of slow disk I/O. As a consequence, the non-blocking servers require additional threads to achieve their best performance and the servers with smaller memory footprints should perform better than the servers requiring more memory. Hence, these experiments should tend to favour the servers that share an address space unless contention becomes an issue. An interesting point is to see how the blocking servers perform compared to the non-blocking servers since the blocking servers have a larger memory footprint.

3.12.1 Tuning Knot

Tuning was again performed for the three versions of Knot: knot-c, knot-nb and knot-b. For all servers, the parameters tuned are the number of threads and the number of worker tasks. Additionally, the size of the application cache is tuned for knot-c. Unlike the experiments with 4 GB of memory, the application cache is not warmed for the knot-c experiments with this workload.

3.12.1.1 Knot-c

Table 3.29 shows the results of tuning knot-c after verification. Each row represents a different number of workers from 5 to 200. The columns are separated into four sections with the results for 8000 threads in section one, 10,000 threads in section two, 15,000 threads in section three and 20,000 threads in section four. In each section, the columns correspond to the three different cache sizes being tested, i.e., 10 MB, 100 MB or 200 MB.

The experiments show the best performance is around 150 workers, at least 8000 threads and 10 MB of application cache (1.39). Based on this general vicinity, additional experiments were run and are presented in Table 3.30. Each row represents a different number of workers from 50 to 200. The columns are separated into two sections with the results for 8000 threads in section one and 9000 threads in section two. In each section, the columns correspond to the two different cache sizes being tested, i.e., 10 MB, or 50 MB. This table shows that the best performance occurs with 8000 threads, 100 workers and 50 MB of application cache (1.44), with a peak throughput of 795 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 782 Mbps at 30,000 requests per second. This result represents a decline of approximately 21% at peak and 18% for the sustained throughput over the best results obtained with knot-c for the 1.4 GB experiments.

	Cache size in MB											
	8000 threads			10,000 threads			15,000 threads			20,000 threads		
Workers	10	100	200	10	100	200	10	100	200	10	100	200
5	1.03	0.81	✕	0.94	0.72	✕	0.70	✕	✕	✕	✕	✕
25	1.37	1.13	0.79	1.31	1.07	✕	✕	✕	✕	✕	✕	✕
50	1.38	1.26	0.87	1.37	1.16	✕	1.11	✕	✕	✕	✕	✕
100	1.38	1.30	0.93	1.38	1.22	✕	✕	✕	✕	✕	✕	✕
150	**1.39**	1.26	✕	1.37	1.16	✕	✕	✕	✕	✕	✕	✕
200	1.38	1.25	✕	1.36	✕	✕	✕	✕	✕	✕	✕	✕

Table 3.29: Knot cache initial experiments - .75 GB

	Cache size in MB			
	8000 threads		9000 threads	
Workers	10	50	10	50
50	1.39	1.42	1.38	1.36
100	1.38	**1.44**	1.38	1.42
125	1.39	1.44	1.38	1.37
150	1.38	1.44	1.38	1.36
200	1.38	1.43	1.38	1.35

Table 3.30: Knot cache fine tune experiments - .75 GB

For this workload, knot-c performs better with small cache-sizes; experiments with cache sizes of 100 MB or larger have lower throughput or do not verify. Based on previous experiments, the fact that knot-c performs better with smaller cache-sizes is expected, but the application cache sizes here are very small. Due to increased memory pressure in the system, the application is disk bound. While the best performing knot-c has no I/O wait at either 15,000 or 30,000 requests per second, the lack of I/O wait is actually the result of polling in the application. At peak, the main Knot thread makes 2,408,380 calls to the event mechanism with 87% of the calls returning no events. This polling indicates there is extra CPU time available, which is reasonable given the smaller number of connections. Hence, using a smaller application-cache at the expense of additional CPU time reduces duplication with the file-system cache and is a reasonable trade off. However, there is a point where this tradeoff is no longer beneficial and further reductions in cache size negatively effect performance. For the same configuration with 10 MB of cache, Knot makes only 389,523 calls to the event mechanism with 85% of the calls returning no events. Polling is reduced because additional overheads associated with managing a smaller application-cache decrease the available CPU time.

3.12.1.2 Knot-nb and knot-b

Table 3.31(a) shows the results of tuning knot-nb. Each row in the table represents a different number of workers from 5 to 200 and each column represents a different number of threads from 8000 to 20,000. For these experiments, the best performance occurs at 8000 threads and at least 50 workers (1.38). Based on this general vicinity, additional experiments were run and are presented in Table 3.32(a). For this table, the number of workers are varied from 25 to 200 and the number of threads from 7000 to 9000. For knot-nb, the best performance occurs with 8000 threads and 50 workers (1.37), with a peak throughput of 769 Mbps occurring at 8000 requests per second and with a sustained throughput of around 734 Mbps at 30,000 requests per second. The occasional verification failures with 8000 workers indicate that knot-nb is close to the maximum number of threads (connections) that it can handle for this workload. This result represents a decline of approximately 40% at 15,000 requests per second and 41% for the sustained throughput over the best results obtained with knot-nb for the 1.4 GB experiments. This decline is significant and much larger than the drop in performance experienced by knot-c.

Comparing the performance of knot-nb and knot-c shows a difference from the trend observed in the previous workloads; knot-nb performs worse than knot-c. For their respective best performing configurations, knot-c has 4% higher peak throughput at 15,000 requests per second and 7% higher sustained throughput at 30,000 requests per second. This result is surprising because the reasons that knot-nb has better performance under the previous two workloads still hold true for this workload.

Consider the performance of both servers at 30,000 requests per second with 8000 threads and 100 workers for both and also with 50 MB of application cache for knot-c. First, knot-nb has a smaller memory footprint than knot-c, resulting in a larger average file-system cache, 595 MB versus 534 MB for knot-c. Second, knot-nb uses zero-copy sendfile versus write for knot-c, resulting in less kernel data copying overhead. Based on profiling data, knot-nb spends 0.47% time performing kernel data copying at peak versus 19.18% for knot-c.

Despite these advantages, however, knot-nb is executing less efficiently than knot-c in some important ways. Knot-nb makes 1,325,180 calls to the event mechanism with 86% of these calls returning no events, indicating that the main knot thread is spending a reasonable amount of time spinning with no other work to do. On the other hand, knot-c makes 852,058 calls to the event mechanism with 77% of these calls returning no events. As well, knot-nb has an average of 328,815 kernel context-switches per second versus 28,474 for knot-c. This additional spinning also explains why knot-nb has higher user-time compared to knot-c, 25% for knot-nb versus 17% for knot-c.

Workers	Number of Threads			
	8000	10,000	15,000	20,000
5	×	×	×	×
25	1.36	×	×	×
50	**1.38**	×	×	×
100	1.38	×	×	×
150	1.37	×	×	×
200	1.37	×	×	×

(a) non-blocking sendfile

Workers	Number of Threads			
	8000	10,000	15,000	20,000
5	×	×	×	×
25	1.36	1.37	1.19	1.04
50	1.43	**1.56**	1.40	1.26
100	1.43	1.56	1.41	1.28
150	1.43	1.55	1.40	1.30
200	1.43	1.55	1.41	1.31

(b) blocking sendfile

Table 3.31: Knot sendfile initial experiments - .75 GB

Workers	Number of Threads		
	7000	8000	9000
25	1.26	1.36	×
50	1.27	**1.37**	×
100	1.27	1.37	×
125	1.27	×	×
150	1.27	×	×
200	1.26	1.37	×

(a) non-blocking sendfile

Workers	Number of Threads		
	9000	10,000	12,000
50	1.55	**1.57**	1.50
100	1.55	1.57	1.50
150	1.55	1.56	1.48
200	1.54	1.57	1.47

(b) blocking sendfile

Table 3.32: Knot sendfile fine tune experiments - .75 GB

This data suggests that knot-nb is spending a large amount of time polling for additional work, in both the main thread and worker tasks, but is unable to take advantage of unused capacity in the system. The usual technique when a server is underutilized is to increase the number of connections and thereby increase the throughput of the server. However, Tables 3.31(a) and 3.32(a) show that increasing the number of connections leads to verification failures for knot-nb. Furthermore, even with additional overheads, knot-c is able to outperform knot-nb with similar configuration parameters.

One advantage of knot-c over knot-nb is the way the two servers read data from disk. Knot-c reads an entire file from disk into its application cache in a single system call before sending any data to the client. This method of disk access is efficient, especially for files that are laid out contiguously on disk. Once the file is in its application cache, knot-c transmits the file to the requestor using write. While multiple calls to write may be necessary if the file is sent in chunks, no further disk I/O is required for that request.

Knot-nb uses non-blocking sendfile to transmit a file to the requestor. With non-blocking sendfile, large files may be transmitted in chunks to the requestor, requiring multiple calls to sendfile in order to complete the transfer. For file data that is not in the file-system cache, a call to sendfile can result in both disk I/O and network I/O. Furthermore, a single call to sendfile may not cause the entire file to be loaded into the file-system cache, meaning that large files requiring multiple sendfile calls likely have to block waiting for disk I/O multiple times. Since each disk I/O request must be queued, the total amount of time it takes to transmit a file increases as multiple calls to sendfile for the same request may block waiting for disk I/O, leading to more large-file timeouts. As well, once the disk head is positioned, it is more efficient to read in data contiguously than jumping around reading in portions of various files. Therefore, knot-c is able to support more threads than knot-nb without verification problems. Based on vmstat output, knot-nb reads in an average of 14,258 blocks per second while knot-c reads in an average of 17,629 blocks per second at 30,000 requests per second.

Table 3.31(b) shows the results of tuning knot-b after verification. Each row in the table represents a different number of workers from 5 to 200 and each column represents a different number of threads from 8000 to 20,000. For these experiments, the best performance occurs at around 10,000 threads and 50 workers (1.56). Based on this general vicinity, additional experiments were run and are presented in Table 3.32(b). For this table, the number of workers are varied from 50 to 200 and the number of threads from 9000 to 12,000. The best performance occurs with 10,000 threads and 50 workers (1.57), with a peak throughput of 888 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 842 Mbps at 30,000 requests per second. This result represents a decline of approximately 30% at 15,000 requests per second and 32% for the sustained throughput over the best results obtained with knot-b for the 1.4 GB experiments. This drop in performance is less than knot-nb but more than knot-c.

For their best configurations, knot-b has 12% better performance than knot-c at 15,000 requests per second and 8% better performance at 30,000 requests per second. As well, knot-b has 15% better performance than the best knot-nb configuration at 15,000 requests per second and 15% better performance at 30,000 requests per second. At 30,000 requests per second, knot-b has a file-system cache size of 574 MB, putting it between knot-c and knot-nb. Compared to knot-c, knot-b manages better performance because of a smaller memory footprint and less overhead due to sendfile. Its use of blocking sendfile allows knot-b to support more threads without verification problems and a larger file-system cache results in higher throughput. On the other hand, knot-nb also uses sendfile and has approximately the same size file-system cache for equivalent experiments but knot-b has better performance. Knot-b outperforms knot-nb because it blocks a worker task until the entire file is sent, giving priority to larger files, resulting in fewer verification problems due to large-file timeouts. As the experiments show, fewer verification problems mean that knot-b can support a larger number of threads (connections), allowing it to achieve better performance. While performance eventually decreases as the number of threads increases, knot-b does not suffer from verification problems like knot-nb. However, even for configuration parameters where knot-nb verifies, knot-b has better performance. There appears to be a performance advantage when using blocking sendfile for this workload.

Based on vmstat output, for its best configuration, knot-b spends 13% of its time waiting for I/O at 30,000 requests per second and 6% at 15,000 requests per second. Increasing the number of workers eliminates I/O wait but does not improve performance. Neither knot-nb nor knot-c have I/O wait for their best configurations, however, given that Knot constantly polls it is difficult to draw any conclusions based on this observation.

3.12.2 Tuning μserver

Experiments were run to tune the four versions of μserver: symped-nb, symped-b, sharedsymped-nb and sharedsymped-b. For all versions of μserver, the parameters tuned are the maximum number of simultaneous connections and the number of processes.

3.12.2.1 Symped-nb and symped-b

Table 3.33(a) shows the results of tuning symped-nb after verification. Each row in the table represents a different number of processes from 5 to 150 and each column a different maximum number of connections from 8000 to 15,000. The experiments show the best performance for symped-nb is around 25 processes and 8000 maximum number of connections (1.24). Based on this general vicinity, additional symped-nb experiments were run and are presented in Table 3.34(a). For this table, the number of processes are varied from 10 to 150 and the maximum number of connections from 7000 to 9000. Table 3.34(a) shows that the best performance for symped-nb occurs with 25 processes and 8000 connections (1.24), with a peak throughput of 680 Mbps occurring at 12,500 requests per second and with a sustained throughput of around 659 Mbps at 30,000 requests per second. This result represents a decline of approximately 48% at 15,000 requests per second and 48% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

The performance of symped-nb is lower than all the Knot servers in the previous section. Similar to knot-nb, symped-nb experiences verification problems when the maximum number of connections is larger than 9000. As well, the performance of symped-nb peaks with 25 processes, unlike the Knot servers which peak with 50–100 workers.

With 25 processes and 8000 connections at 30,000 requests per second, symped-nb has an average file-system cache of 555 MB. This cache size is approximately 40 MB smaller than the average cache-size

Processes	Maximum Number of Connections		
	8000	10,000	15,000
5	1.08	×	×
10	1.19	×	×
25	**1.24**	×	×
50	1.19	×	×
100	×	×	×
150	×	×	×

(a) non-blocking sendfile

Processes	Maximum Number of Connections			
	8000	10,000	15,000	20,000
5	×	×	×	×
10	0.84	0.80	×	×
25	1.23	1.26	1.18	1.08
50	1.29	**1.33**	1.27	1.20
100	1.25	1.26	1.21	1.14
150	1.20	1.19	1.11	×

(b) blocking sendfile

Table 3.33: *µ*server SYMPED initial experiments - .75 GB

Processes	Maximum Number of Connections		
	7000	8000	9000
10	1.15	1.18	1.17
15	1.18	1.22	1.22
25	1.19	**1.24**	1.24
50	1.17	1.20	×
100	×	×	×
125	×	×	×
150	×	×	×

(a) non-blocking sendfile

Processes	Maximum Number of Connections		
	9000	10,000	12,000
35	1.31	1.32	1.31
50	1.32	**1.34**	1.31
100	1.28	1.28	1.26
125	1.25	1.25	1.22
150	1.21	1.20	1.17

(b) blocking sendfile

Table 3.34: *µ*server SYMPED fine tune experiments - .75 GB

of knot-nb with 50 workers and 8000 threads. Increasing the number of processes or connections with symped-nb leads to verification problems, restricting the performance of symped-nb. The limitation on the number of processes is a significant problem given the amount of disk I/O required for this workload because additional processes allow the server to continue servicing requests when a process blocks waiting for disk I/O. If the server cannot support a sufficient number of processes, then CPU time is wasted while the server is I/O blocked; the symped-nb server spends 40% of its time waiting for I/O.

The problem with symped-nb is that its memory footprint grows non-trivially as processes are added since the processes are independent, unlike the Knot servers. For example, moving to 50 processes and 8000 connections at 30,000 requests per second results in an average file-system cache of size 519 MB, a reduction of approximately 36 MB. As the file-system cache size decreases, the number of requests requiring disk I/O increases, eventually resulting in large-file timeouts and verification problems. Therefore, despite unused CPU time available in the system, symped-nb cannot support a large enough number of processes to improve performance.

Table 3.33(b) shows the results of tuning symped-b after verification. Each row in the table represents a different number of processes from 5 to 150 and each column a different maximum number of connections from 8000 to 20,000. The experiments show the best performance for symped-b is around 50 processes and 10,000 maximum number of connections (1.33). Based on this general vicinity, additional symped-b experiments were run and are presented in Table 3.34(b). For this table, the number of processes are varied from 35 to 150 and the maximum number of connections from 9000 to 12,000. Table 3.34(b) shows that the best performance for symped-nb occurs with 50 processes and 10,000 connections (1.34), with a peak throughput of 743 Mbps occurring at 12,500 requests per second and with a sustained throughput of around 700 Mbps at 30,000 requests per second. This result represents a decline of approximately 42% at 15,000 requests per second and 42% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

Table 3.33 shows that symped-b verifies over a larger range of parameters than symped-nb. Not only does the best performing symped-b experiment have more connections and more processes than symped-nb but it also has better performance. Symped-b has 8% better performance than symped-nb at peak and 6% better performance at 30,000 requests per second. However, the performance of symped-b is lower than the Knot servers in the previous section.

The problem with symped-b is its large memory footprint. While symped-nb and symped-b have comparable memory footprints for equivalent configuration parameters, for its best configuration symped-b has an average file-system cache-size of 502 MB at 30,000 requests per second, 53 MB smaller than symped-nb. While a smaller file-system cache likely increases the amount of time a process must wait for disk I/O, a larger number of processes and connections allows the server to service more requests. Overall, the I/O wait is still high at 35% but less than the I/O wait for symped-nb despite having a smaller file-system cache. The net effect is that symped-b has better performance than symped-nb despite having a larger memory-footprint due to more connections and processes.

In general, the problem with the SYMPED μserver is a large memory-footprint; moving to shared-SYMPED reduces the memory footprint of the server and should improve performance. With μserver, the size of the cache table is based on the number of files so the cache-table size is fixed for the various workloads tested but other statically allocated data structures in the server are based on the number of connections. Given the small maximum connections values for these experiments, moving to a shared cache-table should have a noticeable effect on performance.

3.12.2.2 Sharedsymped-nb and sharedsymped-b

Table 3.35(a) shows the results of tuning sharedsymped-nb after verification. Each row in the table represents a different number of processes from 5 to 150 and each column a different maximum number of connections from 8000 to 15,000. The experiments show the best performance for sharedsymped-nb is around 50 processes and 10,000 maximum number of connections (1.45). Based on this general vicinity, additional sharedsymped-nb experiments were run and are presented in Table 3.36(a). For this table, the number of processes are varied from 15 to 150 and the maximum number of connections from 8000 to 12,000. Table 3.36(a) shows that the best performance for sharedsymped-nb occurs with 50 processes and 10,000 connections (1.46), with a peak throughput of 812 Mbps occurring at 8000 requests per second and with a sustained throughput of around 791 Mbps at 30,000 requests per second. This result represents a decline of approximately 42% at 15,000 requests per second and 42% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

The performance of sharedsymped-nb is 19% higher at peak than symped-nb and 20% higher at 30,000 requests per second, due to a smaller memory footprint than symped-nb as a result of the shared cache-table; for its best configuration at 30,000 requests per second, its average file-system cache-size is 604 MB, approximately 49 MB larger than symped-nb. Considering that the system is running with 768 MB of memory, 604 MB is a large amount of space for the file-system cache. The incremental cost of adding additional processes is smaller with shared-SYMPED than with SYMPED. For example, moving from 25 process to 50 process with 8000 connections at 30,000 requests per second decreases the average file-system cache size by 12 MB, approximately one third of the reduction experienced by symped-nb. Since sharedsymped-nb runs with a larger file-system cache, it can support more processes and connections without verification problems, resulting in higher throughput. The larger number of processes and connections for its best configuration allows sharedsymped-nb to utilize the extra CPU time available when a process blocks for disk I/O in order to improve performance. Sharedsymped-nb spends 31% of its time blocked waiting for disk I/O, which is lower than both the SYMPED servers.

Sharedsymped-nb also has a performance advantage over knot-nb, with 6% better performance at peak and 8% better performance at 30,000 requests per second. For their best configurations at 30,000 requests per second, sharedsymped-nb has a larger file-system cache by approximately 9 MB. This difference is not large enough to account for the performance disparity between the two servers. However, at its best configuration knot-nb is only running with 8,000 threads (connections) as it has verification problems for larger connections values, while sharedsymped-nb is running with 10,000 connections. Therefore, sharedsymped-nb supports more connections with a smaller memory footprint, resulting in higher throughput than expected just based on the difference in file-system cache-size. Adding a small number of additional worker tasks has little effect on knot-nb's memory footprint but adding a large number of

	Maximum Number of Connections		
Processes	8000	10,000	15,000
5	1.13	×	×
10	1.27	1.28	×
25	1.37	1.44	×
50	1.40	**1.45**	×
100	1.40	×	×
150	1.40	×	×

(a) non-blocking sendfile

	Maximum Number of Connections		
Processes	8000	10,000	15,000
5	×	×	×
10	×	×	×
25	1.29	1.39	1.31
50	1.41	1.57	1.56
100	1.45	**1.61**	1.59
150	1.46	1.59	1.50

(b) blocking sendfile

Table 3.35: μserver shared-SYMPED initial experiments - .75 GB

	Maximum Number of Connections			
Processes	8000	9000	10,000	12,000
15	1.32	1.36	1.36	×
25	1.36	1.41	1.44	×
50	1.40	1.45	**1.46**	×
100	1.40	1.44	×	×
150	1.40	1.42	×	×

(a) non-blocking sendfile

	Maximum Number of Connections		
Processes	9000	10,000	12,000
35	1.46	1.51	1.53
50	1.52	1.59	1.61
100	1.56	1.63	**1.64**
125	1.57	1.62	1.63
150	1.56	1.61	1.62

(b) blocking sendfile

Table 3.36: μserver shared-SYMPED fine tune experiments - .75 GB

additional threads (connections) has a larger effect. With 50 processes and 8000 connections at 30,000 requests per second, sharedsymped-nb has a file-system cache size of 620 MB, approximately 25 MB larger than knot-nb with equivalent parameters, explaining sharedsymped-nb's slightly better performance even at 8,000 connections.

Table 3.35(b) shows the results of tuning sharedsymped-b after verification. Each row in the table represents a different number of processes from 5 to 150 and each column a different maximum number of connections from 8000 to 15,000. The experiments show the best performance for sharedsymped-b is around 100 processes and 10,000 maximum number of connections (1.61). Based on this general vicinity, additional sharedsymped-b experiments were run and are presented in Table 3.36(b). For this table, the number of processes are varied from 35 to 150 and the maximum number of connections from 9000 to 12,000. Table 3.36(b) shows that the best performance for sharedsymped-b occurs with 100 processes and 12,000 connections (1.64), with a peak throughput of 909 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 879 Mbps at 30,000 requests per second. This result represents a decline of approximately 30% at 15,000 requests per second and 30% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

The performance of sharedsymped-b is 12% higher at peak than sharedsymped-nb and 11% higher at 30,000 requests per second. While sharedsymped-nb and sharedsymped-b have similar memory footprints for equivalent configuration parameters, for its best configuration at 30,000 requests per second, sharedsymped-b has a file-system cache-size of 561 MB, approximately 43 MB smaller than sharedsymped-nb for its best configuration. Similar to knot-b and symped-b, despite having a smaller memory footprint sharedsymped-b performs better than sharedsymped-nb.

Sharedsymped-b also has 2% better performance than knot-b at peak and 4% better performance at 30,000 requests per second. For their best configurations at 30,000 requests per second, knot-b's file-system cache is 14 MB larger than sharedsymped-b but its performance is lower. Similar to the non-blocking version, sharedsymped-b supports more connections than knot-b, for their best performing configurations. Staying with 50 workers but increasing to 12,000 threads, knot-b's file-system cache is 552 MB at 30,000 requests per second, 9 MB smaller than sharedsymped-b with equivalent parameters. Similarly, sharedsymped-b has 22% better performance than symped-b at peak and 26% better performance at 30,000 requests per second. In this case, for their best configurations at 30,000 requests per second, sharedsymped-b has a file-system cache that is 59 MB larger while supporting 2000 additional connections, resulting in better overall performance. Around the vicinity where best performance occurs, sharedsymped-b has a smaller memory footprint than an equivalently configured knot-b or symped-b, resulting in better performance. As well, a smaller memory footprint allows sharedsymped-b to support a larger number of connections, resulting in further performance improvements.

For its best configuration at 30,000 requests per second, sharedsymped-b spends 20% of its time blocked waiting for disk I/O. While this value is lower than the I/O wait for the other servers discussed so far for this workload, it is still not zero. Sharedsymped-b does not suffer from verification problems, so it is possible to examine the effect of increasing the number of connections. Increasing the number of connections to 15,000 results in performance decreasing to 863 Mbps and I/O wait increasing to 25% because the average file-system cache shrinks by 27 MB. By decreasing the size of the file-system cache, the number of requests requiring disk I/O increases to a point where increasing the number of connections no longer yields performance benefits but hurts performance. At this point, increasing the number of processes to compensate for the additional connections only exacerbates the problem by further shrinking the file-system cache.

3.12.3 Tuning WatPipe

Table 3.37(a) shows the results of tuning watpipe-nb after verification. Each row in the table represents a different number of writers from 5 to 150 and each column a different maximum number of connections from 8000 to 15,000. The experiments show the best performance for watpipe-nb is around 100 writers and 10,000 maximum number of connections (1.52). Based on this general vicinity, additional watpipe-nb experiments were run and are presented in Table 3.38(a). For this table, the number of writers are varied from 15 to 200 and the maximum number of connections from 9000 to 12,000. Table 3.38(a) shows that the best performance for watpipe-nb occurs with 100 writers and 10,000 connections (1.52), with a peak throughput of 880 Mbps occurring at 8000 requests per second and with a sustained throughput of around 823 Mbps at 30,000 requests per second. This result represents a decline of approximately 40% at 15,000 requests per second and 40% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

The performance of watpipe-nb is 9% higher at peak than sharedsymped-nb and 4% higher at 30,000 requests per second. However, for its best configuration at 30,000 requests per second watpipe-nb has an average file-system cache size of 590 MB, which is 14 MB smaller than sharedsymped-nb. Watpipe-nb verifies over a larger range of configuration parameters than sharedsymped-nb, with more stable performance across these parameters. The reason for watpipe-nb's stability is that its tasks share an address space so the incremental cost of additional writers is small. Initially, watpipe-nb has a larger memory footprint than sharedsymped-nb but the memory footprint of sharedsymped-nb increases faster as processes are added compared to watpipe-nb as writer tasks are added. Both sharedsymped-nb and watpipe-nb have approximately the same performance with 50 workers and 10,000 connections. Sharedsymped-nb has a file-system cache that is 13 MB larger at 30,000 requests per second. With watpipe-nb, moving from 50 writers to 100 writers and 10,000 connections at 30,000 requests per second results in the file-system cache shrinking by approximately 1 MB. With sharedsymped-nb, moving from 50 processes to 100 processes and 10,000 connections at 30,000 requests per second results in the file-system cache shrinking by approximately 22 MB. Hence, watpipe-nb has a larger file-system cache with 100 writer tasks versus sharedsymped-nb with 100 processes, both at 10,000 maximum connections. Since watpipe-nb supports more writer tasks with less memory overhead, it is able to realize a performance advantage by moving to 100 writer tasks.

Watpipe-nb spends 26% of its time waiting for disk I/O with 100 writers and 10,000 connections at 30,000 requests per second, lower than the other non-blocking servers examined. I/O wait represents an opportunity to improve performance by taking advantage of unused CPU time. While watpipe-nb does not have verification problems over the range of tuning values tested, it is unable to improve performance and eliminate I/O wait. The reason is that its memory footprint grows slowly as writer tasks are added

Writers	Maximum Number of Connections		
	8000	10,000	15,000
5	1.03	0.99	0.93
10	1.21	1.19	1.10
25	1.35	1.37	1.29
50	1.40	1.48	1.38
100	1.41	**1.52**	1.44
150	1.42	1.52	1.44

(a) non-blocking sendfile

Writers	Maximum Number of Connections		
	8000	10,000	15,000
5	×	×	×
10	0.87	0.79	0.69
25	1.37	1.42	1.26
50	1.43	1.66	1.56
100	1.43	1.71	1.68
150	1.43	**1.72**	1.68

(b) blocking sendfile

Table 3.37: WatPipe initial experiments - .75 GB

Writers	Maximum Number of Connections		
	9000	10,000	12,000
15	1.30	1.28	1.25
25	1.40	1.39	1.34
50	1.46	1.46	1.43
100	1.51	**1.52**	1.49
150	1.51	1.52	1.50
200	1.51	1.52	1.48

(a) non-blocking sendfile

Writers	Maximum Number of Connections		
	9000	10,000	12,000
35	1.56	1.58	1.54
50	1.60	1.66	1.65
100	1.61	1.72	**1.77**
125	1.60	1.72	1.74
150	1.60	1.72	1.75
200	1.60	1.71	1.74

(b) blocking sendfile

Table 3.38: WatPipe fine tune experiments - .75 GB

and grows at about the same rate as sharedsymped-nb as connections are added. At 10,000 connections, adding additional processes beyond 100 does not improve performance though I/O wait does decline. Moving to 12,000 connections reduces the file-system cache-size by 14 MB, increases I/O wait to 30% and reduces performance. Increasing the number of writers to 150 reduces I/O wait to 25% and improves performance slightly but not back to the level of its best performance. Further increasing the number of writers decreases I/O wait, but does not improve performance even though the file-system cache-size decreases only slightly. The disk is clearly the bottleneck inhibiting performance.

Table 3.37(b) shows the results of tuning watpipe-b after verification. Each row in the table represents a different number of writers from 5 to 150 and each column a different maximum number of connections from 8000 to 15,000. The experiments show the best performance for watpipe-b is around 150 writers and 10,000 maximum number of connections (1.72). Based on this general vicinity, additional watpipe-b experiments were run and are presented in Table 3.38(b). For this table, the number of writers are varied from 35 to 200 and the maximum number of connections from 9000 to 12,000. Table 3.38(b) shows that the best performance for watpipe-b occurs with 100 writers and 12,000 connections (1.77), with a peak throughput of 1002 Mbps occurring at 15,000 requests per second and with a sustained throughput of around 968 Mbps at 30,000 requests per second. This result represents a decline of approximately 27% at 15,000 requests per second and 27% for the sustained throughput over the best results obtained for the 1.4 GB experiments.

The performance of watpipe-b is 14% higher at peak than watpipe-nb and 18% higher at 30,000 requests per second. For its best configuration at 30,000 requests per second, watpipe-b has an average file-system cache-size of 577 MB, which is 13 MB smaller than watpipe-nb. Again, the WatPipe servers have similar memory footprints for equivalent configuration parameters, so the memory difference is due to differing parameters for their best configurations. For all the servers examined, the blocking version has better performance than its non-blocking counterpart with this workload.

The performance of watpipe-b is 10% higher at peak than sharedsymped-b and 10% higher at 30,000 requests per second. For its best configuration at 30,000 requests per second, watpipe-b has an average file-system cache that is 16 MB larger than sharedsymped-b. This difference is not large enough to explain the performance disparity between the servers since both watpipe-b and sharedsymped-b have their best performance with the same configuration parameters. The performance difference between the servers arises because of factors other than memory footprint. While the previous discussion has shown a strong correlation between throughput and memory footprint for servers with equivalent sendfile behaviour, a memory difference of less than 3% cannot account for a 10% difference in performance.

A couple of other factors also contribute to watpipe-b having better performance than sharedsymped-b. Watpipe-b spends 16% of its time waiting for disk I/O, which is the smallest amount of time among all the servers tested, versus 20% for sharedsymped-b. One advantage that watpipe has over shared-SYMPED is that its tasks perform different actions. With sharedsymped-b, it is possible for all the processes associated with the server to become blocked waiting for disk I/O. In this case, extra CPU time that could be devoted to accepting new connections, reading new requests or polling is wasted. With watpipe-b, when all the writer tasks block waiting for disk I/O, the remaining tasks continue to process other stages of the pipeline. As the number of connections are fixed, at some point the remaining tasks must wait for disk I/O to complete before they can proceed, resulting in a low but non-zero I/O wait. Similar to the other servers, at some point the trade off between the increase in memory footprint from additional connections outweighs the performance gains and the performance of the server declines. However, watpipe-b is able to push that point further than the other servers due to its architecture. As well, WatPipe has lower costs associated with event polling than shared-SYMPED because WatPipe has centralized polling whereas each shared-SYMPED process performs its own event polling (see OProfile data in Section 3.13).

3.13 Server Comparison

Figure 3.8 presents the best performing configuration for each server-architecture implementation: caching Knot (knot-c), non-blocking Knot (knot-nb), blocking Knot (knot-b), *μ*server non-blocking SYMPED (symped-nb), *μ*server blocking SYMPED (symped-b), *μ*server non-blocking shared-SYMPED (sharedsymped-nb), *μ*server blocking shared-SYMPED (sharedsymped-b), non-blocking WatPipe (watpipe-nb) and blocking WatPipe (watpipe-b). The legend in Figure 3.8 is ordered from the best performing server at the top to the worst at the bottom. Peak server performance varies by about 47% (680–1002 Mbps).

Table 3.39 ranks the performance of the servers for the .75 GB workload. Again, based on a total of three runs for each server, Tukey's Honest Significant Difference test is used to differentiate the servers with a 95% confidence level. The servers are then ranked based on mean area.

The top performer is watpipe-b, with performance 10% better than the next best server, sharedsymped-b. Third are knot-b and watpipe-nb, which is the highest performing non-blocking sendfile server. The next grouping consists of knot-c and sharedsymped-nb with approximately the same performance. Finally, come knot-nb and the SYMPED servers with knot-nb and symped-b having better performance than symped-nb.

Comparing the performance of the best version of WatPipe and the best version of *μ*server, watpipe-b has a 10% higher peak than sharedsymped-b at 15,000 requests per second and 10% higher performance after saturation at 30,000 requests per second. Comparing the performance of the best version of WatPipe and the best version of Knot, watpipe-b has a 13% higher peak than knot-b at 15,000 requests per second and 15% higher performance at 30,000 requests per second. For this workload, knot-c does not have the worst performance; it places in the middle of the servers tested. However, compared to knot-c, knot-b has a 12% higher peak at 15,000 requests per second and 8% higher performance at 30,000 requests per second. The best blocking server, watpipe-b compared to the best non-blocking server, watpipe-nb, has a 14% higher peak at 15,000 requests per second and 18% higher performance at 30,000 requests per second. Between the best blocking *μ*server and the worst non-blocking *μ*server versions is a larger gap; sharedsymped-b has a 34% higher peak at 15,000 requests per second and 33% higher performance at 30,000 requests per second compared to symped-nb. Symped-nb is the worst performing server, with symped-b having a 9% higher peak at 15,000 requests per second and 6% higher performance at 30,000 requests per second.

To better understand the performance of the servers, the best configuration of each server is profiled. The OProfile and vmstat data for these experiments are summarized in Tables 3.40 and 3.41. Additional vmstat data is presented for this workload; the row labelled "blocks in" gives the average number of blocks read in per second. Note, a non-zero I/O wait value indicates that the profiling data must be scaled because

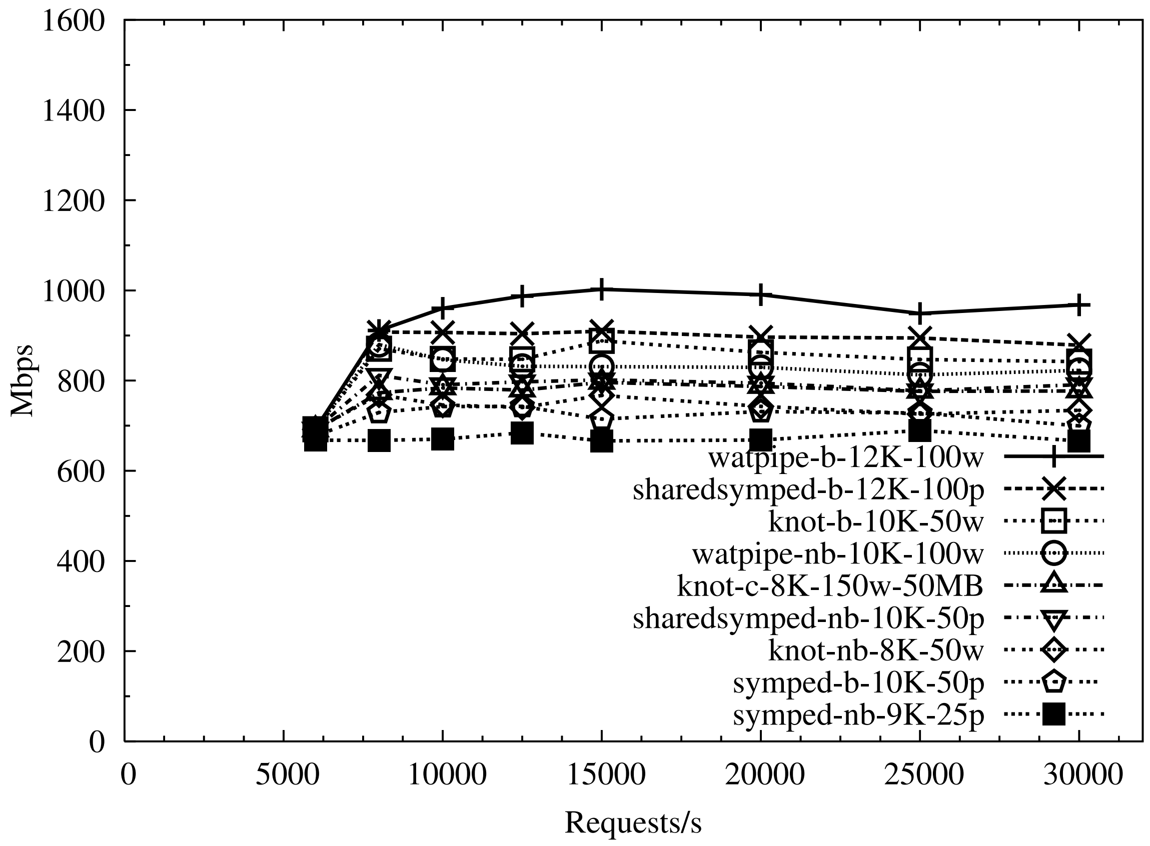


Figure 3.8: Throughput of different architectures - .75 GB

Server	Rank
watpipe-b	1
sharedsymped-b	2
knot-b	3
watpipe-nb	3
knot-c	4
sharedsymped-nb	4
knot-nb	5
symped-b	5
symped-nb	6

Table 3.39: Ranking of server performance - .75 GB

the profiling data gathered 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 that the CPU is in use.

The SYMPED and shared-SYMPED servers have lower user-space execution totals than the other servers by 50% or more. The relative user-space totals for μserver have stayed consistent over the various workloads but the totals for Knot and WatPipe are highest for this workload. With Knot, both the main thread and the worker tasks poll for additional work. Since there is extra CPU time as a result of waiting for disk I/O, Knot spends this time polling, leading to increased application totals compared to the other servers. With WatPipe, the various threads in the system are not busy enough to keep running so there is a lot of blocking and signaling of threads, resulting in higher thread overheads. As μserver is event-driven it does not suffer from these overheads.

However, the downside of the SYMPED and shared-SYMPED servers is high kernel event-poll overheads. Since all the μserver processes are symmetric, each process independently calls the event mechanism. For example, sharedsymped-b makes 2945 calls per second to epoll_wait, while watpipe-nb with asymmetric tasks has less overhead as it only makes 1745 calls per second.

The SYMPED and shared-SYMPED servers appear to have high networking and e1000 values but not correspondingly high throughput. However, scaling the networking and e1000 values based on the appropriate I/O wait shows that these values are consistent with the other servers.

Similar to the other workloads tested, knot-c has large data-copy overhead. As well, additional overheads related to user-level threading still exist for all the Knot servers. For this workload, however, these overheads do not have a large effect on performance. Because the workload is disk bound, there is extra CPU time available in the system to absorb these overheads, reducing their effect on performance. Knot-nb and knot-b spend any leftover CPU time polling, resulting in a high level of kernel context-switching that results in large scheduling overheads and higher user-space totals. For knot-c, the data-copying overhead required to maintain an application cache consumes this extra CPU time. Note the low kernel context-switching and scheduling overhead values for knot-c. Hence, knot-c's performance penalty for not using sendfile is offset and it also gains some efficiency with respect to disk access by performing large contiguous disk reads into the application cache. The result is that the performance of the Knot servers are similar to the other servers despite additional overheads.

The discussion in the previous sections showed a correlation between the file-system cache-size and the throughput of a server. For equivalent configurations, the server with a smaller memory footprint has higher throughput. Furthermore, a smaller memory footprint often allows the server to achieve its best performance with a larger number of processes and/or connections, resulting in additional performance improvements. Since the servers are disk bound, the expectation is that the average throughput of the disk is approximately the same for all the servers and the size of the file-system cache only effects the number of requests that must block waiting for disk I/O. However, according to the vmstat data in Tables 3.40 and 3.41, the blocks-in per second values show a large variation across the servers.

For analysis, divide the servers in two separate groups: non-blocking sendfile servers and blocking sendfile servers. The difference in blocks-in among the servers appears to be related to two factors, the number of kernel threads (workers, processes or writers depending on the server) and the size of the file-system cache. If the number of kernel threads is kept the same, then servers with larger file-system caches have higher blocks-in per second and higher throughput. Since the system is under memory pressure, the most frequently requested files are likely already in the file-system cache, while the remaining files need to be read in from disk when requested. Due to the Zipf distribution of requests, smaller files are requested more frequently than larger files. Therefore, as the size of the file-system cache increases, the average size of a file serviced from disk also increases. Requesting larger files from disk means that on average more contiguous data is read on each disk read, so disk efficiency increases and the amount of data that can be read from disk also increases. The servers with 100 kernel threads have file-system cache-sizes that are less than 20 MB smaller than the best performing servers with 50 kernel threads. For these small differences, having more kernel threads is an advantage as these additional threads are able to keep the disk more consistently busy, resulting in higher blocks-in. However, since the file-system cache continues to shrink as kernel threads are added, the average file-size of requests requiring disk I/O becomes smaller and disk efficiency eventually drops resulting in lower performance.

When all the servers are analysed together, a performance anomaly emerges when comparing blocking versus non-blocking sendfile servers. As mentioned in the sections discussing the performance of individual servers for this workload, the blocking version of a server has better performance than the equivalent non-blocking version. This behaviour is unusual because in all cases, for their best configurations, the blocking server has a smaller file-system cache than its non-blocking counterpart. For some of the servers, part of the performance difference can be attributed to the non-blocking servers encountering verification problems at the configuration parameters where the blocking version of the server performed best. However, the blocking servers even outperformed the non-blocking servers at the best configuration parameters for the non-blocking server. Furthermore, watpipe-nb does not suffer from verification problems but watpipe-b still has better performance. Since the only difference between the WatPipe servers is blocking versus non-blocking sendfile, for equivalent configuration parameters their file-system cache sizes are approximately the same and cannot account for this performance difference. One unexpected difference that is consistent among the servers is that the blocking version of the server has higher blocks-in per second than the non-blocking version of the server. Since the speed of the disk remains consistent and the disk is a bottleneck for this workload, the expectation is that all the servers are maximizing disk I/O and the average throughput of the disk is consistent among the servers assuming that other factors are equivalent. But the blocks-in per second values for the servers suggest that the blocking servers are able to get more data from disk than the non-blocking servers.

Read access to each file for all the servers is sequential. However, with non-blocking sendfile the access pattern for reading large files may be misinterpreted by the kernel. With sendfile, the data is transmitted in pieces, with the size of each piece determined by the socket-buffer size. Similarly, the operating system reads a file into the file-system cache from disk in pieces, with the size of each piece determined by the I/O scheduling algorithm. For large files, the socket-buffer size is smaller than the amount of file-data read in by a single disk request, so the number of disk accesses required is fewer than the number of network transmissions required for a request of a large file that is not already in the file-system cache. With non-blocking sendfile, the file-access pattern appears random because subsequent sendfile calls do not appear to continue from the end of the last disk I/O. At this point, the kernel disables page-caching read-ahead for the file and the size of the disk requests become smaller on average. With blocking sendfile, only a single sendfile call is required, and since the kernel performs the appropriate tracking, it recognizes that file access is sequential, resulting in two benefits. First, the average disk-request size is larger, resulting in better disk efficiency. Second, since the blocking server has page-caching read-ahead enabled, the kernel requests the next section of the file be read in from disk while it is still processing the current section of the file. Hence, the total amount of time to send an entire file is reduced as the transmission of the file is overlapped with the reading of the file from disk, leading to fewer large-file timeouts. The misclassifying of the non-blocking sendfile access-pattern is likely a deficiency of the Linux kernel used for these experiments and may not apply to newer kernels or other operating systems.

Once the servers become disk bound, there is extra CPU capacity in the system so execution efficiency becomes less important than disk-access efficiency when comparing servers and server architectures. Knot-c's performance with this workload is an example of the importance of disk efficiency. For this experiment, the two factors affecting disk efficiency are blocking versus non-blocking sendfile and memory footprint. The use of blocking sendfile made the biggest difference, with the blocking servers outperforming the non-blocking servers despite having smaller memory footprints. However, once the servers are divided based on sendfile, small memory footprints result in better performance. For this workload, the servers with at least some shared address-space performed better, with the two SYMPED servers having the worst performance and no shared address-space.

Server Arch Write Sockets	**Knot-cache** **T/Conn** **non-block**	**Knot** **T/Conn** **non-block**	**Knot** **T/Conn** **block**	**userver** **symped** **non-block**	**userver** **symped** **block**
Max Conns Workers/Procs/Writers Other Config	**8K** **100w** **50MB**	**8K** **50w**	**10K** **50w**	**8K** **25p**	**10K** **50p**
Reply rate Tput (Mbps)	6470 770	6283 750	6974 832	5384 641	5881 700
OPROFILE DATA					
vmlinux total %	**65.97**	**59.45**	**60.11**	**62.45**	**63.25**
networking	19.31	18.35	22.32	26.41	26.14
memory-mgmt	7.35	4.81	5.52	6.84	6.89
file system	4.25	3.41	3.79	5.38	5.55
kernel+arch	5.53	5.72	6.33	9.72	9.76
epoll overhead	1.36	1.53	1.69	4.65	4.63
data copying	19.18	0.47	0.53	0.95	0.93
sched overhead	2.04	10.91	8.03	1.02	1.57
others	6.95	14.25	11.90	7.48	7.78
e1000 total %	**15.69**	**15.61**	**17.79**	**23.79**	**22.35**
user-space total %	**15.72**	**22.56**	**19.62**	**7.34**	**7.07**
thread overhead	7.03	10.97	9.68	0.00	0.00
event overhead	0.00	0.00	0.00	2.54	2.37
application	8.69	11.59	9.94	4.8	4.7
libc total %	**0.02**	**0.02**	**0.02**	**3.60**	**4.36**
other total %	**2.60**	**2.36**	**2.46**	**2.82**	**2.97**
VMSTAT DATA					
waiting %	0	1	6	39	31
file-system cache (MB)	528	593	570	542	478
blocks-in/sec	17,101	14,514	16,489	13,670	16,066
ctx-sw/sec (kernel)	9984	319,285	194,558	4575	7264
SERVER STATS					
ctx-sw/sec (user)	16,012	14,506	13,794		

Table 3.40: Server performance statistics gathered under a load of 15,000 requests per second - .75 GB

Server Arch Write Sockets	**userver** **s-symped** **non-block**	**userver** **s-symped** **block**	**WatPipe** **pipeline** **non-block**	**WatPipe** **pipeline** **block**
Max Conns Workers/Procs/Writers Other Config	**10K** **50p**	**12K** **100p**	**10K** **100w**	**12K** **100w**
Reply rate Tput (Mbps)	6502 778	7586 908	7068 843	8322 990
OPROFILE DATA				
vmlinux total %	**63.00**	**64.95**	**61.35**	**62.26**
networking	26.02	25.26	20.79	23.01
memory-mgmt	7.04	6.74	6.02	6.21
file system	4.37	4.26	3.61	3.95
kernel+arch	10.33	11.05	12.97	11.79
epoll overhead	5.05	5.85	2.56	2.88
data copying	0.97	0.93	0.67	0.71
sched overhead	1.84	3.13	5.48	4.71
others	7.38	7.73	9.25	9.00
e1000 total %	**23.45**	**21.25**	**18.84**	**18.41**
user-space total %	**7.45**	**7.29**	**15.38**	**14.96**
thread overhead	0	0	10.35	9.78
event overhead	2.63	2.45	1.90	1.89
application	4.82	4.84	3.13	3.29
libc total %	**3.37**	**3.85**	**1.65**	**1.67**
other total %	**2.73**	**2.66**	**2.78**	**2.70**
VMSTAT DATA				
waiting %	30	13	23	10
file-system cache (MB)	598	554	585	572
blocks-in/sec	15,352	18,643	16,605	19,501
ctx-sw/sec (kernel)	7933	14,967	21,278	19,977
SERVER STATS				
ctx-sw/sec (user)				

Table 3.41: Server performance statistics gathered under a load of 15,000 requests per second - .75 GB

3.14 Comparison Across Workloads

The previous sections concentrate on understanding the performance of the various servers under three different workloads. This section examines the performance of the servers across the workloads. As the workload changes, the general behaviour of the servers are similar; as memory pressure increases, the throughput of the servers decreases. However, the relative performance of the servers is not consistent over the three workloads.

Figure 3.9 graphs the throughput of servers versus the system memory size across the three workloads at 15,000 requests per second. The knot-c experiments across the workloads show that using sendfile is crucial for high performance. Using write results in high data-copying overheads and cache duplication, resulting in lower throughput. The performance of knot-c has already been discussed in the previous sections, so knot-c is ignored in the following discussion, except where referred to explicitly.

The servers achieve their best performance with the 4 GB workload, when there is no memory pressure in the system. At this point, differences in architecture, aside from memory footprint, have the largest effect on the performance of a server. For the in-memory workload, servers requiring few kernel threads tend to have low overheads and high throughput. Since the non-blocking servers require only a few kernel threads, as no overlapping of CPU and disk I/O can occur since all data reads (disk I/O) occur without blocking, the non-blocking servers perform better than the blocking servers. The blocking servers have higher overheads as they require a large number of kernel threads to handle blocking network I/O. The additional overheads associated with user-level threading result in lower performance for the Knot servers. However, unlike with other comparisons of server architectures for in-memory workloads, the differences in performance among the various architectures are small and the performance of all the servers is reasonable.

When the entire file set no longer fits into the file-system cache, some requests require disk I/O to complete. If the system is able to overlap computation with disk I/O, eliminating the time a server is entirely blocked waiting for I/O, the drop in performance is small. Once the amount the memory pressure in the system is large, disk I/O becomes the bottleneck and the throughput of the servers drops sharply as the servers begin blocking waiting for disk I/O to complete. A reasonable expectation is that server performance should tend to converge as the memory pressure in the system increases and the speed of disk I/O becomes the major bottleneck. However, server throughput does not converge.

The most important factors governing performance are the memory footprint and the disk efficiency of the server. For servers with similar disk efficiency, once there is memory pressure in the system, the memory footprint of the server dictates performance; a smaller memory footprint results in higher throughput. As discussed in section 3.13, the blocking sendfile servers have better disk efficiency than the non-blocking sendfile servers, though that is at least partly due to the kernel read-ahead problem. The

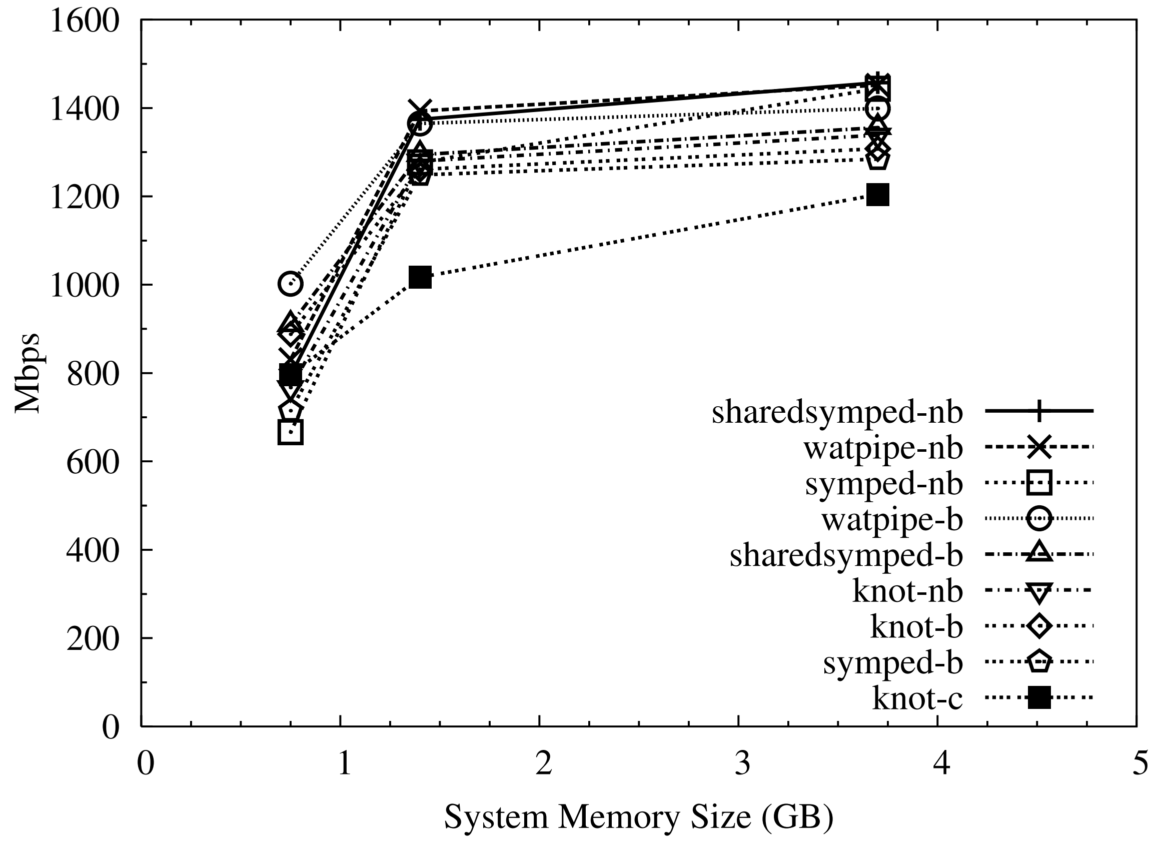


Figure 3.9: Comparison of server throughput at 15,000 requests per second across workloads

effect of these two factors can be observed in both the 1.4 GB and .75 GB workloads. Other factors such as maximum connections and number of kernel threads are also important to achieve high throughput, but to a large extent these factors are constrained by the memory footprint of the server.

The throughput of all the servers decreases when moving from the 4 GB workload to the 1.4 GB workload. However, the throughput of the non-blocking servers decreases more than the blocking servers, resulting in a tighter grouping of server performance. The larger drop in throughput is partially because of the overhead of the non-blocking servers going from one or few kernel threads to many kernel threads, incurring more overheads; e.g., when *μ*server goes from SPED to SYMPED with multiple processes. However, the blocking servers also have the advantage of better disk efficiency, resulting in smaller drops in throughput. As a result, the performance of the blocking servers is close to the performance of the non-blocking servers despite having larger memory footprints and more overheads since additional kernel threads are required. Since the disk is not saturated and I/O wait can be eliminated, the effect of disk efficiency is lower for this workload compared to the .75 GB workload. In fact, ignoring knot-c, the spread of throughput for the servers is smallest for the 1.4 GB workload.

The relative performance of the blocking servers continues to improve compared to the non-blocking servers for the .75 GB workload. Since the workload is disk bound, I/O wait cannot be eliminated and disk efficiency becomes more important. As the amount of disk I/O increases, disk efficiency becomes crucial for high throughput and so the blocking servers begin performing better than the non-blocking servers. Again, for servers with similar disk efficiency, a server with a larger file-system cache has higher throughput. Therefore, the servers that share address space perform better than the SYMPED servers. The disk efficiency of the blocking servers and the large differences in memory footprint relative to the available memory result in the largest spread in throughput for the .75 GB workload.

The tuning parameters to achieve best performance varies across the servers but they tend to behave consistently across the workloads. The expectation is that the number of kernel threads required to achieve best performance increases with the memory pressure in the system. For the non-blocking sendfile servers, one kernel thread is required for the in-memory workload, fifteen to twenty-five for the 1.4 GB workload and twenty-five to one hundred for the .75 GB workload. The range of kernel threads gets larger as the level of disk I/O increases because the increase in memory footprint as additional workers/processes/writers are added constrain the maximum number of kernel threads to achieve best performance. Therefore, servers with shared address-space are able to support a larger number of kernel threads. For the blocking sendfile servers, seventy-five to one hundred kernel threads are required for the in-memory workload, seventy-five for the 1.4 GB workload and fifty to one hundred for the .75 GB workload. When the server is not disk bound, the number of kernel threads required is dictated by the throughput of the servers. Since the servers have higher throughput with the in-memory workload, more requests tend to block on sendfile, so more kernel threads are required. Once the servers are disk bound, the memory footprint of the server constrains the number of kernel threads.

For the second tuning parameter, number of connections, the relationship is more straightforward. The throughput of the server dictates the maximum number of connections it can support without verification problems. As the amount of disk I/O increases, the throughput of the servers decrease and best performance occurs with fewer maximum connections. As discussed previously, the blocking servers can support a larger number of connections without verification problems and achieve their best performance with a larger number of connections.

3.15 Summary

This chapter examines the performance of several server architectures on a uniprocessor machine for a spectrum of workloads, from in-memory to disk bound. The architectures include thread-per-connection, SYMPED, shared-SYMPED and pipeline. For the first three architectures, an existing high performance server implementation is chosen, Knot running on top of the Capriccio thread library for the thread-per-connection server and *μ*server for SYMPED and shared-SYMPED. For the last architecture, WatPipe, a new pipeline server is implemented. Previous research has shown both Knot and *μ*server to be among the best for their respective architectures and WatPipe has comparable performance. Significant effort has been undertaken to make the servers consistent, allowing for a fair comparison. As well, the servers employ newer operating system features, i.e., sendfile and edge-triggered epoll, to reduce overhead and improve throughput. The servers have been extensively tuned to find the best performing configuration and both a non-blocking sendfile and blocking sendfile version of each server is examined. In addition, a version of Knot using an application file-cache with write is also tested. This version of Knot is similar to the original version, except it uses epoll for its event mechanism and has some modifications to make it consistent with the other servers, e.g., updated hashing code.

The experiments show that tuning the server parameters is critical to achieving high throughput. In some cases there is a narrow window of best tuning, especially for the non-blocking servers, and missing that window can result in large performance penalties. Limiting the maximum number of connections for non-blocking servers is important to maintain high throughput after saturation. One important problem with the non-blocking servers is large-file timeouts. As the blocking servers focus on replying to current requests before reading new requests, they tend to be immune to large-file timeouts. This behaviour is also self-limiting, so the blocking servers are less sensitive to a larger maximum number of connections. With respect to the number of kernel threads, performance of the servers with shared memory and asymmetric tasks is stable as the number of kernel threads increase. Even when there is no memory pressure in the system, the servers with symmetric processes show less tuning stability as the number of kernel threads increase due to additional overheads.

The servers are verified to ensure that all file sizes are equally serviced. Typically, once the tuning parameters exceed a certain value, either performance levels off, performance decreases or verification failures occur. However, examining the throughput of experiments failing verification shows the performance of the server levels off or decreases in this case as well. This observation is reasonable because if a server is attempting to equally service all file sizes, verification problems indicate timeouts for requests in progress, resulting in wasted processing time.

One big difference between the servers is how additional kernel threads are introduced into the server. *μ*server uses symmetric processes, while the other servers use asymmetric kernel threads, Knot has a main thread and worker tasks and WatPipe has threads dedicated to servicing the various stages. The advantage of symmetric processes (or kernel threads) is that scheduling is easier. The fair scheduling of the default operating system scheduler does a reasonable job for symmetric processes. On the other hand, scheduling asymmetric threads can be difficult. Fairly scheduling all threads may be inappropriate, forcing the application to self schedule its threads. In WatPipe, for example, it is unreasonable to allow an event polling thread to repeatedly call the event mechanism without allowing the other threads in the application to process the available events between calls. This coordination introduces additional overheads, can be tricky and may not be portable. However, asymmetric threads also offer advantages over symmetric threads. One advantage of asymmetric threads is they allow adding threads where needed, without affecting overheads for other parts of the application. For example, adding additional worker tasks in Knot to handle disk I/O. While neither approach shows significant advantages for the servers tested, the symmetric approach seems to work better when fewer kernel threads are required and the asymmetric approach when a larger number of kernel threads are required. For example, the servers with asymmetric threads have a smaller performance difference between their non-blocking and blocking versions than the servers with symmetric processes, for the in-memory workload.

The experiments show the throughput of the servers is affected by the workload. The servers have highest throughput with the 4 GB workload, followed by the 1.4 GB workload and finally the .75 GB workload. Since servicing requests from memory is faster than servicing requests from disk, the amount of disk I/O required dictates the maximum throughput of the server. All the architectures tested are a viable choice for a high-performance web-server. New operating system facilities, such as sendfile, epoll and efficient context switching bring the performance of the servers together. However, as the workload changes, the relative performance of the servers also change; no single server performs the best across all the workloads tested. For the 4 GB workload, *μ*server non-blocking shared-SYMPED, *μ*server non-blocking SYMPED and non-blocking WatPipe have the highest throughput at 15,000 requests per second, approximately 9% higher than non-blocking Knot. For the 1.4 GB workload, non-blocking WatPipe and non-blocking shared-SYMPED have the highest throughput at 15,000 requests per second, approximately 9% higher than both non-blocking SYMPED and non-blocking Knot. For the .75 GB workload, blocking WatPipe has the highest throughput at 15,000 requests per second, approximately 10% higher than blocking shared-SYMPED and 13% higher than blocking Knot and 35% higher than blocking SYMPED. The additional overheads related to user-level threading with a large number of threads for Knot result in it consistently performing lower than the other servers.

Once there is memory pressure in the system, the biggest factors determining the performance of a server are memory footprint and disk efficiency. For the disk bound workload, the blocking servers perform better than the corresponding non-blocking servers despite having a larger memory footprint because of better disk efficiency. Overall, blocking WatPipe has the best performance. It benefits from both better disk efficiency from blocking sendfile and from a smaller memory footprint due to its completely shared address space once there is memory pressure in the system. If a system is under memory pressure, the biggest factor in determining whether an architecture is reasonable appears to be memory footprint. It may be possible to improve the memory footprint of all the servers tested, resulting in better performance.

Based on the experiments in this chapter, no single server performs best across all the workloads; the peak throughput difference among the best version of each server is within 9–13%, depending on the workload, indicating that all the servers perform well. If achieving maximum throughput is critical, non-blocking WatPipe and non-blocking shared-SYMPED offer the best performance for in-memory and moderate disk-I/O, and blocking WatPipe offers the best performance when there is heavy disk-I/O. Knot has extra overheads related to supporting a large number of user-threads. Non-blocking *μ*server SYMPED performs well for in-memory workloads but *μ*server SYMPED does not scale efficiently as the number of processes increase because the processes have completely separate address-spaces.

Given the reasonable performance exhibited by all the servers, factors other than throughput may be more important in deciding which server architecture is appropriate. For example, the experiments show tuning is important to achieve the best performance but the best tuning for a server varies based on workload. Therefore, servers that are less sensitivity to larger tuning parameters are easier to tune and a single tuning may be reasonable for a wider range of loads. The servers with shared memory and asymmetric tasks are easier to tune because they show less tuning sensitivity across all workloads as the number of kernel threads are increased. As well, the blocking servers have fewer verification problems as the maximum number of connections increase. Based on these additional criteria, blocking WatPipe is a good server choice. Blocking WatPipe performs well across the workloads, i.e., it has peak throughput within 4% of the best server for the in-memory workload, 2% for the moderate disk I/O workload and 10% higher peak throughput for the disk-bound workload. Furthermore, it is easier to tune than the other servers; its performance is stable as the number of writer tasks is increased and it does not have verification problems as the number of connections is increased.

Chapter 4

Multiprocessor Web-Server Architectures

This chapter examines various server architectures run on a multiprocessor with different workloads. The servers from Chapter 3, aside from Knot, are extended for multiprocessor execution and both the N-copy version of the uniprocessor server and the extended version of the server are tested. Two workloads are explored: the first is in-memory and the second is disk bound. Similar to the uniprocessor experiments, the varying workloads are achieved by reconfiguring the server with different amounts of memory, while keeping other factors the same.

For the multiprocessor experiments, partitioning the subnets and processes/kernel-threads is important to achieve high performance (see Section 4.4). While achieving this partitioning is straightforward for the N-copy servers, the server architectures are extended to allow this partitioning as well. The goal of this chapter is to compare the performance of the server architectures on a multiprocessor and to show that performance comparable to N-copy can be achieved with the extended servers. Again, each server is individually tuned for best performance on each workload and the best configuration of each server is profiled to examine any differences and similarities among the architectures, and for different implementations within an architecture.

4.1 Overview

The methodology for the experiments in this chapter is generally the same as the methodology used in Chapter 3. Specifically, httperf is run with a 10 second timeout value, the same verification procedures are followed and similar tuning is performed on each the servers. However, based on experience gained with the uniprocessor experiments, only a single round of tuning is performed. For the experiments in this chapter, the request rates range from 25,000 requests per second to 70,000 requests per second. Differences in file set and environment are discussed further in the next two sections.

Before running experiments to measure the performance of the various servers, some preliminary experiments are run. These preliminary experiments determine the affinity settings required to maximize performance and examine the scalability of the hardware and operating system.

Performance experiments are run to test each server on two workloads. The two workloads are labelled based on the size of the server memory: 4 GB and 2 GB. These workloads correspond to in-memory and heavy disk I/O. Note, 2 GB of system memory results in heavy disk-I/O for these experiments due to higher throughput resulting from multiple processors. For each workload, a number of tuning experiments are run for each server.

Additional data is gathered for each experiment in three ways. Similar to the uniprocessor experiments, vmstat is run with a 5 second interval and each server gathers statistics while it is running. In addition, mpstat is run during the experiments. The mpstat utility is run with a 5 second interval to sample the system state and generate per CPU data about where CPU-execution time is spent. It breaks down the information into categories such as user time, system time, software interrupts (softirqs), interrupts per second, etc. Similar to the uniprocessor experiments, only the condensed area under the throughput curve for each experiment is reported. The remainder of the data is not included in the thesis, but a summary of some of the data gathered is presented when necessary to provide further explanations.

Finally, no Knot experiments are run for this chapter. Knot, when running on the Capriccio thread library, only supports multiple kernel-threads for writing so it can only be run N-copy. Given this limitation, Knot is uninteresting on a multiprocessor so it is not considered further.

4.2 File Set

The file set used for the experiments in this chapter is static and generated using the SPECweb99 file-set generator. Similar to the file set used in Chapter 3, 650 directories were generated, resulting in approximately 3.1 GB of files on the server. Based on preliminary performance experiments, the 650 directories are distributed over the server's two hard drives to achieve higher throughput when disk I/O occurs (not done in the uniprocessor experiments). The list of the files in each directory and their profile is the same as the files in Chapter 3.

The multiprocessor experiments are run with 16 clients (versus 8 in the uniprocessor experiments), each running a copy of the httperf load generator, requiring a new set of 16 log files, also with requests conforming to a Zipf distribution, to account for the additional clients. As discussed earlier, the actual file set for the experiments is based on the files present in the client log-files. Due to the way the log files are generated, not all the generated files on the server are used. The client log-files request 2.2 gigabytes, consisting of 21,600 files across all 650 directories of the file set. Hence, the actual size of the file set is approximately 2.2 GB. For the experiments in this chapter, the log file for each client has an average session length of 7.29 and all the requests are for static files. Like Chapter 3, the various workloads for this chapter are generated by keeping the file set and client log files constant and reconfiguring the server with different amounts of memory.

Table 4.1 shows the amount of memory required to satisfy requests as the file size increases. The distribution of requests over the file sizes is very similar to the log files in Chapter 3 but not exactly the same. Again, due to the Zipf distribution, only a small amount of memory is needed to service a significant percent of the requests; 50 percent of the requests comprise 8.4 MB of the file set and 95 percent of the requests comprise only 126.5 MB of the file set.

% Reqs	Memory Size (MB)	Max File size (B)
10	0.5	409
20	0.8	512
30	1.5	716
40	4.8	3072
50	8.4	4096
60	9.9	5120
70	12.2	5120
80	20.1	7168
90	94.3	40,960
95	126.5	51,200
100	2291.6	921,600

Table 4.1: Cumulative amount of memory required for requests when sorted by file size

4.3 Environment

The experimental environment consists of eight client machines and a single server. The client machines are identical to the clients in Chapter 3. The server machine contains one quad-core E5440 2.83 GHz Xeon CPU, 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 one add-on card and the remaining two from another add-on card. For all experiments, the server runs a 2.6.24-3 Linux kernel in 32-bit mode with SMP enabled. Switching to a newer kernel is required to properly support the hardware on the server machine. With a 32-bit address space, the default configuration for the Linux kernel is to assign 1 GB of virtual address space for the kernel and 3 GB for user processes. Using the default 1 GB/3 GB memory split, the kernel began to run out of available memory for some experiments due to a large number of processes and sockets. The problem appears to be related to slow reuse of kernel memory, eventually resulting in insufficient free memory for new connections over the course of an experiment. In order to accommodate experiments with a large number of processes and sockets, the kernel has been compiled to use a 2 GB/2 GB memory split between the kernel and user-space.

The 2.6.24-3 kernel already contains the fix for the caching problem discovered in the 2.6.16-18 kernel used in the previous chapter. However, the kernel code path for sendfile is different in the 2.6.24-3 kernel versus the 2.6.16-18 kernel, so the fix is not relevant for the experiments in this chapter. Furthermore, the read-ahead code has also been extensively modified and does not exhibit the problem seen in Chapter 3 (see Section 3.13 for details). Unfortunately, I discovered that the new code path has a page caching problem that results in poor disk performance. When sendfile is used to transmit a file to a client, none of the pages associated with the file are marked as accessed by the kernel. Not marking page-accesses means the kernel cannot distinguish between recently or frequently accessed pages and other pages in the file-system cache. Therefore, under memory pressure the kernel evicts random pages from the file-system cache. The problem is these random pages could include pages in the middle of files, frequently accessed pages, etc., resulting in erratic performance and low disk-throughput as long contiguous disk reads and read-ahead buffering are less useful as disk requests become smaller and more random. Therefore, the experiments in this chapter use a 2.6.24-3 kernel that I patched so that page accesses are correctly marked for sendfile calls (see Section A.2 for patch). Disk throughput improved from approximately 11,000 blocks in per second for non-blocking sendfile and 20,000 blocks-in for blocking sendfile to approximately 28,000–30,000 blocks-in for both non-blocking and blocking sendfile with the patch. As well, the difference in disk performance between blocking and non-blocking sendfile is small and the variation in throughput is low for repeated experiments. Due to improved disk throughput, the performance of the servers also improved with the patch.

The connection between the server and client machines is the same as that described in section 3.5, except scaled to 8 client machines. Again, each client machine has two CPUs and runs two copies of the workload generator, resulting in 16 clients evenly spread over eight network interfaces on the server. The clients, server, network interfaces and switches have been sufficiently provisioned to ensure that the network and clients are not the bottleneck.

4.4 Affinities

On a single processor, all the processes and interrupt servicing must execute on that processor. When multiple processors exist, there are several options regarding how the execution of processes and interrupt servicing can be distributed. Of particular interest for these experiments is the execution of the kernel threads associated with the server and the interrupt servicing for the network interfaces.

The default behaviour is to allow the system to schedule the server kernel-threads and interrupt requests (IRQs). In order to control the behaviour of the kernel with respect to IRQ scheduling, the Linux kernel used in this chapter is built with IRQ_BALANCE disabled because the IRQ balancing functionality in the Linux kernel does not perform well. Other simple strategies like handling interrupts round-robin among the CPUs also perform poorly due to poor cache behaviour. Instead, IRQ scheduling should be configured manually to achieve best performance for specific server hardware. The Linux kernel also allows setting CPU affinities for kernel threads and for interrupts. Setting CPU affinity ties a kernel thread or interrupt to a specific processor or set of processors.

The server machine in the test environment has multiple processors and multiple network-interfaces, so a number of configurations exist. Affinities can be set independently for the network interfaces and for the server processes. This section considers some of the options for an N-copy server using non-blocking sendfile. For these experiments, μserver SYMPED is used. Based on Chapter 3, one μserver process per CPU should be sufficient and achieve reasonable performance.

There are four CPUs, so four copies of the symped-nb server are executed. In order for the servers to cover all eight subnets, each symped-nb server handles two distinct subnets with no subnet handled by more than one server. This configuration results in a unique association of servers and subnets that allows segregation of server processes and network interfaces. In Figure 4.1, the same experiment is chosen for all configurations: 4 processes with 20,000 maximum connections per process for a total of 80,000 connections. Verification failures are ignored and just the throughput of the server is considered. However, both experiments where the network interface affinities are not set fail verification.

In the first experiment, *no affinity*, no affinities are set to allow subsequent comparisons with various affinity settings. Both the network interfaces and the processes are free to use any CPU. Note the server does not have IRQ_BALANCE enabled in the kernel or installed on the system. The peak throughput is 2375 Mbps at 25,000 requests per second.

Based on the mpstat output for the experiment, one CPU spends 100% of its time servicing interrupts and the other processors spend a small amount of time servicing interrupts, 5% or less on average. Note the processor that becomes dedicated to handling interrupts does vary between experiments but is constant during the run for each individual rate. However, a single processor handling the majority of interrupts is a

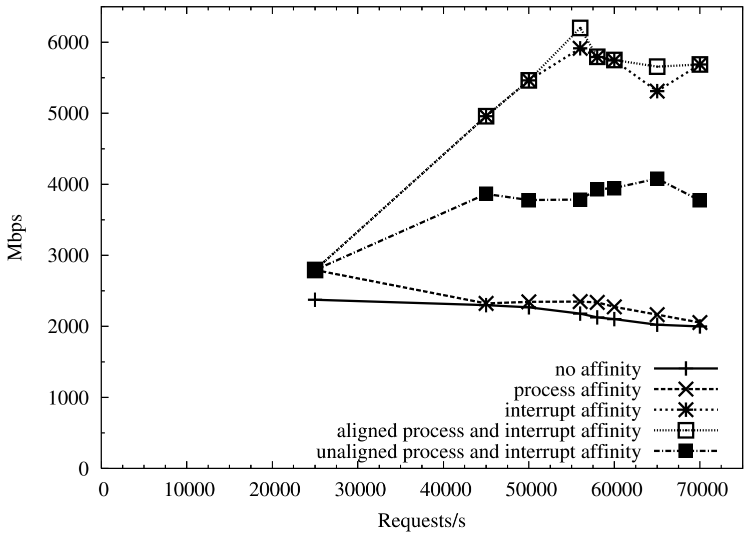


Figure 4.1: *μ*server N-copy non-blocking with 4 processes, 80,000 connections and various affinities

bottleneck because it does not have enough CPU capacity to handle interrupt processing for eight network interfaces, inhibiting performance. Given this organization, the server processes receive insufficient work, resulting in low throughput and a large amount of idle time on the remaining CPUs.

In the second experiment, *process affinity,* each server process has its affinity set to a separate processor and interrupt affinities are not set. Setting affinities for the processes results in a slight improvement, with a peak throughput of 2794 Mbps at 25,000 requests per second. The mpstat output again reveals that the system schedules the servicing of interrupts on a single processor, resulting in a bottleneck as the CPU is quickly saturated. As a result, the server process tied to the saturated CPU is starved for execution time after interrupt servicing occupies 100% of the CPU.

In the third experiment, *interrupt affinity,* interrupt affinities are set for the network interfaces but the processes are free to execute on any of the CPUs. Setting the network-interface affinities results in a significant improvement in performance; the peak throughput is 5915 Mbps at 56,000 requests per second. Examining the mpstat output reveals that the average software interrupt servicing time, as well as, user time, system time, etc. is approximately the same across all the CPUs for each rate. However, Table 4.2 shows that performance can be inconsistent across experiments; the condensed area at 80,000 connections is lower than the value for 60,000 and 100,000 connections. This anomaly occurs for the following reason. By default, the Linux scheduler schedules a particular process consistently on the same CPU unless a load imbalance occurs. When it detects a load imbalance, a process is selected to execute on another CPU. So when the server processes are not bound, they tend to execute on a single CPU and this CPU is likely the same as the CPU tied to servicing its interrupts. However, when this coincidental affinity does not occur,

	Maximum Number of Connections			
Procs	40,000	60,000	80,000	100,000
4	12.87	16.75	16.62	16.76

Table 4.2: Experiments with only network interface affinities set (condensed area)

	Maximum Number of Connections			
Procs	40,000	60,000	80,000	100,000
4	12.87	16.84	16.85	16.84

Table 4.3: Experiments with aligned network interface and process affinities set (condensed area)

there can be poor performance at that rate, in this case 65,000 requests per second. (Note the drop in the *interrupt affinity* line at that point in Figure 4.1). Based on a number of experiments (not shown), the instability can randomly occur at different request rates with various configuration parameters. This type of instability occurs as a result of not binding each processes to a specific CPU.

In the fourth experiment, *aligned process and interrupt affinity*, both process and interrupt affinities are set and aligned to correspond so that the server process handling requests from a particular subnet is bound to the same CPU as the CPU servicing interrupts for that subnet. The peak throughput is 6202 Mbps at 56,000 requests per second. Unlike the previous experiment, setting both process and interrupt affinity results in stable throughput across all the experiments, see Table 4.3.

Finally, a fifth experiment, *unaligned process and interrupt affinity*, is run to confirm that the alignment of interrupt servicing and processes to the same CPU is important. In this case, both process and interrupt affinities are set but the alignment is explicitly set so that processes and network interfaces are unaligned. The performance of this experiment is much lower than the interrupt affinity and aligned process and interrupt affinity experiments. In fact, the fifth experiment did not pass verification. Ignoring verification problems, however, the peak throughput for the unaligned experiment is 4081 Mbps at 65,000 requests per second, compared to 6202 Mbps at 56000 requests per second for the aligned process and interrupt affinity experiment. In this case, after the experiment has been executing for a while two of the CPUs become mostly dedicated to servicing interrupts and the server processes on those CPUs become starved for execution time. As those processes are only handling a small number of requests, the time spent servicing interrupts on the other two remaining CPUs drops. Effectively, only two of the processes are servicing requests and the other two processes are starved. Clearly, aligning the processes and network interfaces results in the best performance. In addition, this experiment confirms that the Linux scheduler does a reasonable job of scheduling the processes based on the interrupt affinity settings with this type of workload. While the Linux scheduler does a reasonable job if the process affinities are not set, it is better to align the affinities, if possible, to guarantee the best performance.

The term partitioned is used to describe the general case of a server running with aligned process and network affinities. Specifically, partitioning means that the kernel threads processing a request must execute on the same CPU that handles the network interrupt processing for the subnet associated with the request. For certain experiments in this chapter, steps are taken in order to achieve partitioning. First, affinities are set so that network interrupt processing for a subnet occurs on a single CPU. For load balancing, interrupt processing for the eight subnets are equally distributed over the four CPUs, so each CPU handles interrupt processing for two distinct subnets. Second, each kernel thread in a server only processes requests from subnets associated with the a particular CPU, and affinities are set so the kernel thread only executes on that CPU. As will be seen with WatPipe in Section 4.6.3, absolute partitioning of all the kernel threads is not always required.

Other work has also shown that aligning processor and interrupt affinities yields the best performance [4, 24]. Foong *et al.* [24] performed similar experiments to test TCP performance under various affinity settings. Their experiments also show that aligning process and interrupt affinities yields the best performance. In addition, they also provide an explanation as to why only setting interrupt affinities performs almost as well as aligning process and interrupt affinities. According to their explanation, tasklets, deferrable functions related to interrupt handling, are usually scheduled on the same processor as the interrupt handler, indirectly resulting in aligned process affinity. As well, they provide a detailed analysis explaining why aligned process and interrupt affinities result in the best performance. They note improvements in cache misses, pipeline flushes and locking across various parts of the TCP pipeline.

Anand and Hartner [4] examined TCP/IP performance on the 2.4 and 2.5 versions of the Linux kernel. They showed that aligning both process and interrupt affinities results in better data and instruction locality, resulting in better cache performance and higher throughput.

4.5 Scalability

In order to achieve higher throughput, the simplest approach is to have multiple, identically configured single-processor machines each running an independent copy of a web server. Assuming that an external mechanism exists to handle load balancing, the performance of the system should scale perfectly from one to N machines. An alternative approach that may seem similar is to run independent copies of a web server on a multiprocessor system, with one server copy per CPU. Despite the copies of the server running independently, sharing at the operating system and hardware level can have a significant effect on scalability.

Experiments were run to examine the scalability of running μserver N-copy non-blocking SYMPED on a multiprocessor machine. The purpose of the experiment is to determine if running on a multiprocessor scales similarly to running on separate single-processor machines. To accomplish this comparison, in-memory experiments were run as both the server hardware (CPUs and network interfaces) and client load are scaled proportionately to simulate independent single-processor machines. To perform the equivalent uniprocessor experiment, the server is booted with 1 CPU running a single copy of *µ*server with four clients generating requests over two subnets. Correspondingly, if N CPUs are booted, the experiment consists of running N copies of *µ*server with 4 $\times$ N clients generating requests, simulating N single-processor machines. Experiments with N equal to 1, 2 and 4 were run.

Each *µ*server process is set up to receive requests from two subnets with no two processes sharing a subnet, i.e., each process communicates exclusively with two network interfaces. Affinities are aligned so a process and the interrupts for its corresponding network interfaces are handled by the same CPU and no two server processes share a CPU. The idea is to minimize sharing of resources among copies of the server. As well, the experiments are setup so that each *µ*server process receives the same sequence of requests; i.e., the same set of log files are used for each group of clients associated with a *µ*server copy. Since the number of clients are scaled along with the CPUs, aggregate request rates are also scaled so that the request rate for each individual client remains the same. For example, with 1 CPU, an aggregate request rate of 8,000 requests per second would result in each of the four clients running at a rate of 2000 requests per second. With 4 CPUs, the aggregate request rate is adjusted to 32,000, resulting in each of the sixteen clients running at a rate of 2000 requests per second.

Unfortunately, when running with 1 CPU, the single copy of *µ*server achieves line speed on the two network interfaces before the CPU is fully utilized. While utilizing additional network interfaces would have solved this problem, scaling to 4 CPUs would require more client machines than available and more network interfaces than the server machine can support. To mitigate this problem, special measures are taken for the scalability experiments. First, the speed of the CPUs is reduced from 2.83 GHz to 2.00 GHz. Second, transmit and receive checksumming is disabled on the network interfaces to increase the amount work done by the CPUs. These measures reduced the throughput enough so the scalability experiments could be run with the existing hardware setup.

Table 4.4 contains the results of the scalability experiments. Perfect scalability is not achieved. With 4 CPUs, the throughput is around 2.3 times the throughput with a single CPU. Interestingly, the speedup from 2 to 4 CPUs is also 1.5. The table shows that the request rate at which the servers peak is lower (after appropriate scaling) than the expected perfect scaling. For example, with 1 CPU the server peaks at 14,000 requests per second but with 2 CPUs the server peaks at 22,500 requests per second, not 28,000 requests per second.

To compare the servers, experiments with a single consistent rate, scaled for each configuration, are examined, see Table 4.5. In this case, the rate chosen is 33,000 requests per second, the peak rate for 4

CPUs	Rate	Throughput (Mbps)	Speedup	Idle Time (%)
1	14,000	1556	1	4
2	22,500	2367	1.5	1
4	33,000	3603	2.3	2

Table 4.4: Scalability of μserver N-copy SYMPED

CPUs, with scaled rates of 16,500 for 2 CPUs and 8250 for 1 CPU. As expected, the scalability at these rates is almost perfect since virtually all the requests are successfully handled. Only the 4 copy version had a small number of client timeouts. More important for these experiments is to examine the average idle time, based on vmstat output during the experiment. The idle time in Table 4.5 is averaged over the CPUs, so 2% means 2% per CPU in the 4 CPU case. From 1 CPU to 2 CPUs, the idle time drops by 12% and from 2 CPUs to 4 CPUs the idle time drops by a further 24%. The decrease in idle time as the number of CPUs increases indicates that at some level the servers are not entirely independent, and hence, perfect scaling would not continue at higher request rates.

CPUs	Rate	Throughput (Mbps)	Speedup	Idle Time (%)
1	8250	907	1	38
2	16,500	1814	2	26
4	33,000	3603	4	2

Table 4.5: Scalability of μserver N-copy SYMPED at a consistent rate

For a further breakdown, mpstat output is also gathered during the experiments. The only significant difference in the mpstat values is the time spent in system-level and software-interrupt (softirq) code. The averages with 1 copy: 29% for system, 23% for softirq, with 2 copies: 35% system and 29% softirq and with 4 copies: 55% system and 38% softirq. Again all the times are averaged over the number of CPUs. As load is scaled based on the number of CPUs, ideally all parts of the system should have the same relative performance, so the percent of time spent in each part should remain constant. Given that the user times (not shown) are within 0.5%, μserver is scaling linearly, so the non-linear parts appear to be concentrated in the kernel.

Despite attempting to segregate the server processes and their associated hardware, including CPU and network interfaces, the system does not scale linearly because the operating-system kernel, processor caches and hardware buses are still shared. These elements of the system inhibit parallelism of the server processes. The focus of this chapter is to examine the effect of various server architectures within the confines of these limitations.

Veal and Foong [54] also analysed the scalability of a web server on a multiprocessor. Similar to the experiments in this section, they found scalability problems as the number of cores increased. Based on extensive profiling, they determined that address-bus capacity is the primary bottleneck that inhibited scaling of the web server on eight cores for their machine and environment. However, both the application and kernel also exhibited some scalability problems.

4.6 4 GB

This section considers the performance of various web-server architectures when the entire file set fits into the file-system cache. For these experiments, the system was configured with 4 GB of memory, resulting in 3.5 GB of available memory due to parts of the address space being reserved for hardware devices. Eliminating memory pressure highlights the multiprocessor characteristics of the various architectures without focusing on disk I/O. The next section examines the effect of disk I/O. Note, the special measures taken for the scalability experiments are not in effect for the other experiments in this chapter; the CPU speed has been reset to 2.83 GHz and transmit and receive checksumming on the network interfaces has been re-enabled. Based on the throughput speedup achieved with four processors, the network capacity for the server and test environment are sufficiently provisioned.

The servers can be run in various ways to achieve parallel execution on a multiprocessor. The experiments in this section try to cover some different options for achieving parallel execution but not all possible configurations are tested for each option. One reasonable technique for achieving parallel execution is the N-copy approach discussed in Section 4.4. Aside from N-copy, the servers can be run with multiple kernel threads, similar to the uniprocessor experiments. Based on experiments in Section 4.4, network interrupt affinities are always set, so that interrupt processing is equally distributed across the CPUs, by pinning two network interfaces to each CPU. Where possible, process affinities are also set so that network and process affinities are aligned in order to partition the system. For example, the N-copy experiments in this section segregate the CPU and network interfaces associated with each server copy so the experiments achieve better performance.

4.6.1 Tuning N-copy

As the experiments use static workloads, no communication is required among the various server processes and connections; hence, N-copy is a reasonable architecture. While the SYMPED and shared-SYMPED servers already have an N-copy design, there are some differences between those servers and the N-copy architecture in this section. The SYMPED and shared-SYMPED servers have a single listening port with all processes handling connections from any subnet. This approach does not allow specific processes to be associated with particular subnets and for the hardware to be partitioned based on this association. By running N copies of the SYMPED, shared-SYMPED and pipeline servers, the desired segregation can be achieved. Since there is no communication among the servers and the system is not under memory pressure, the N-copy approach likely represents the best possible performance for the servers.

The problems when running concurrent programs on a multiprocessor are locking, synchronization, cache coherency, etc. By setting affinities appropriately, the idea is to run the server as a number of uniprocessor servers each running on a single CPU. This approach tries to eliminate multiprocessor issues among servers at the application level as the servers are independent and can only be scheduled on a single CPU. It also tries to minimize multiprocessor issues in the kernel by segregating the hardware and servers as much as possible. However, locking and sharing of data structures still occurs within the Linux kernel to control access to shared resources such as the disk, file system, bus, network cards, etc.

Based on uniprocessor tests for non-blocking sendfile, only one kernel thread is required per CPU when there is no disk I/O. Based on testing in Section 4.4, segregating the subnets among the servers and then running the servers on the same CPU as the subnets it is handling is the best approach. Again, since there are eight subnets and four server copies, each server must handle connections from two subnets.

4.6.1.1 Tuning N-copy μserver

Since μserver is already a form of N-copy, a bit of explanation is required to differentiate the experiments in this section. In the N-copy experiments earlier in this chapter, there is only one process per CPU with affinities set to segregate the hardware. Hence, each server copy is essentially a separate instance of a SPED server. For the next set of experiments, each copy of the N-copy server is a general SYMPED or shared-SYMPED server potentially consisting of multiple processes sharing the same subnets and running on a single CPU. The machine used for these experiments has four CPUs available and so for these N-copy experiments, N equals four. Consider an N-copy experiment with a blocking SYMPED server where each copy of the server uses 100 processes. A single copy of the server would consist of 100 processes each sharing two subnets all tied to a single CPU. The entire experiment consists of 400 processes across eight subnets and all four CPUs, i.e., 4 of the single copies.

Experiments were run to tune three versions of μserver: N-copy symped-nb, N-copy symped-b and N-copy sharedsymped-b (but not N-copy sharedsymped-nb, see below). As discussed later in this section, no N-copy sharedsymped-nb experiments were run. For all versions of μserver, the parameters tuned are the maximum number of simultaneous connections and the number of processes.

Table 4.6(a) shows the results of tuning N-copy symped-nb. Each row in the table represents a different number of processes from 4 to 16 with the number of processes representing the total number of processes running. The total number of processes must be divided by four to determine the number of processes for each server copy. For example, the 8 processes in the second row represent 4 symped-nb servers each with 2 processes. The columns represent a different maximum number of connections from 40,000 to 100,000. Again, the columns represent the total number of connections across all the servers. Hence, the entry for 8 processes and 60,000 connections means that each of the four symped-nb server copies has a maximum of 15,000 connections. Furthermore, each individual symped-nb process has a maximum of 7500 connections as each server copy has 2 processes in this example. The experiments show the best performance for N-copy symped-nb is around 4 processes and 100,000 connections (16.87), with a peak throughput of 6202 Mbps occurring at 56,000 requests per second and with a sustained throughput of around 5695 Mbps at 70,000 requests per second. This throughput is substantially larger than those seen in the uniprocessor experiments. However, the results cannot be compared as the hardware and operating systems are different.

One interesting observation for the best configuration is that after the peak the throughput drops by 6.5% from 6204 Mbps to 5800 Mbps at 58,000 requests per second and then starts to level off. While the condensed areas at 60,000 and 100,000 connections are similar, 16.85 and 16.87 respectively, the shape of the throughput curves are different. At 60,000 connections, the peak is 5964 Mbps at 56,000 requests per second but after peak the performance holds steady with a throughput of 5954 Mbps at 58,000 requests per second. Hence, there is a trade off with lower peak throughput but a more gradual decline after peak.

Similar to the uniprocessor experiments, when there is no disk I/O, efficiencies are gained by only having a single process. In this case, a single SPED server on each processor is equivalent and performs the best. As well, the performance of these experiments level off beyond 60,000 maximum connections. N-copy symped-nb running with a single server process per CPU is self-limiting with an average of less than 16,500 concurrent connections per server. Therefore, similar to the uniprocessor SPED experiments, additional connections beyond 100,000 do not result in higher throughput.

Table 4.6(b) shows the results of tuning N-copy symped-b. Each row in the table represents a different number of processes from 200 to 600, with the number of processes representing the total number of processes running. The total number of processes must be divided by four to determine the number of processes for each server copy. For example, the 300 processes in the second row represent 4 symped-b server copies each with 75 processes. The columns represent a different maximum number of connections from 40,000 to 100,000. Again, the columns represent the total number of connections across all the servers. Hence, the entry for 300 processes and 80,000 connections means that each of the 4 symped-b server copies has a maximum of 20,000 connections. Furthermore, each individual symped-b process has a maximum of 267 connections as each server has 75 processes in this example. The experiments show

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.86	16.85	16.86	**16.87**
8	✕	✕	16.68	16.68
12	12.87	✕	16.66	16.64
16	✕	✕	16.64	16.63

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	12.19	14.14	13.21	12.87
300	12.59	15.34	14.52	13.89
400	12.75	**15.54**	15.05	14.66
500	12.55	15.41	15.28	14.95
600	12.42	14.87	15.18	14.74

(b) blocking sendfile

Table 4.6: μserver N-copy SYMPED experiments - 4 GB

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	12.18	13.93	13.45	✕
300	12.64	15.09	14.34	13.58
400	12.73	15.24	14.98	14.39
500	12.81	15.31	15.17	14.94
600	12.91	**15.38**	15.26	15.11

(a) mutex lock

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	12.18	14.22	13.15	12.88
300	12.63	15.46	14.57	13.81
400	12.76	15.77	15.43	14.68
500	12.82	15.82	15.62	15.30
600	12.93	15.79	15.72	✕
700	13.04	**15.88**	15.77	15.69
800	12.91	15.73	15.81	15.68

(b) readers/writer lock

Table 4.7: μserver N-copy blocking shared-SYMPED experiments - 4 GB

the best performance for N-copy symped-b is around 400 processes and 60,000 connections (15.54), with a peak throughput of 5346 Mbps occurring at 50,000 requests per second and with a sustained throughput of around 4286 Mbps at 70,000 requests per second. The throughput is 14% lower at peak than the throughput of the corresponding non-blocking version and 25% lower at 70,000 requests per second.

The problem is that N-copy symped has a large memory footprint. Even for 200 processes and 40,000 connections, the memory footprint of the server is large enough that the file set cannot fit into the remaining memory. For the best performing configuration at peak, the average file-system cache is 1.99 GB, resulting in disk I/O but no I/O wait. Since the amount of disk I/O increases as the number of processes and/or connections increase, the server becomes disk bound and the overall performance of the server decreases, see Table 4.6(b). As seen in Chapter 3, each symped-b server copy requires at least 50 processes for stable performance, hence running with fewer processes reduces performance.

To mitigate the large-memory footprint-problem, some N-copy sharedsymped-b experiments were run. With N-copy sharedsymped-nb, the best performing server is expected to be the SPED version of the server as this is consistent with the experiments in Chapter 3 and the N-copy symped-nb experiments earlier in this section. Since, sharedsymped-nb and symped-nb are the same when the servers are run with a single process, the N-copy sharedsymped-nb experiments are unnecessary.

Table 4.7(a) shows the results of tuning N-copy sharedsymped-b. Each row in the table represents a different number of processes from 200 to 600. The columns represent a different maximum number of connections from 40,000 to 100,000. Again, 4 copies of the shared-SYMPED server are run, with the rows and columns representing the total number of processes and connections across all the servers. The experiments show the best performance for N-copy sharedsymped-b is around 600 processes and 60,000 connections (15.38), with a peak throughput of 5153 Mbps occurring at 50,000 requests per second and with a sustained throughput of around 4444 Mbps at 70,000 requests per second. The throughput is 17% lower at peak than the throughput of N-copy symped-nb and 22% lower at 70,000 requests per second.

A number of the experiments had a higher peak than the experiment yielding the best overall performance but large performance drops after peak result in worse overall performance for these experiments. For example, the experiment with 300 processes and 60,000 connections (15.09) has a higher peak of 5459 Mbps at 50,000 requests per second but its throughput drops by more than 20%, with a throughput of 3706 Mbps at 70,000 requests per second. Similar to the uniprocessor experiments, these drops in throughput occur with blocking servers when there are insufficient processes to handle higher request rates. As well, since the number of workers required is related to throughput, more workers per CPU are needed for these experiments. Additional processes result in a tradeoff because the increase in overhead results in a small decrease in peak throughput but more processes result in a large increase in sustained throughput.

The N-copy sharedsymped-b server has a smaller memory footprint than the corresponding N-copy symped-b server. It is not until 100,000 connections and at least 400 processes that the memory footprint of the N-copy sharedsymped-b server is large enough so that there is not enough memory available for the file-system cache to contain the entire file set. Since most of the experiments do not experience memory pressure, the pattern of performance for the N-copy sharedsymped-b server is more in line with expectations, i.e., performance is stable. Once disk I/O starts to occur, as with N-copy symped-b, the performance of a server drops.

At its best performing configuration, N-copy sharedsymped-b has no idle time at peak for 600 processes. However, the peak throughput for its best configuration is only about 1% higher than its peak for 400 processes and 80,000 connections, where it has an average of 5% idle time at 56,000 requests per second. The larger number of processes improve performance, especially for rates after peak, but at the cost of higher overhead. The difference in idle time is even more pronounced for higher rates. For example, with 400 processes and 80,000 connections at a rate of 70,000 requests per second N-copy, sharedsymped-b has 26% idle time on average. The high amount of idle time combined with the low throughput of 4029 Mbps suggests that the performance of the server is inhibited by lock contention. The only sharing occurring between the processes at the application-level is the cache table, which is analyzed next.

Exclusive access to the cache is only required when an entry is added or removed from the table. As each file is only added to the table once, most cache table accesses are lookups to find existing entries. Multiple cache table lookups can proceed simultaneously as they do not require exclusive access to the cache table. Once an entry is found, exclusive access to that entry is required so its usage information can be updated. If the entry is not found, then the entire cache table must be locked so the new entry can be added to the table. Therefore, the current strategy of locking the entire cache table for every access inhibits concurrency. To test the contention hypothesis, the cache table mutex lock is replaced with a readers/writer lock. The changes are implemented by using two-tiered locking, with a furwock [25] used to lock the cache table and one futex per individual cache entry. The furwock is acquired with exclusive access when entries are added or removed from the cache table. Otherwise, read access is acquired when searching the cache table and then individual cache entries are locked for updating.

Table 4.7(b) shows the results of tuning N-copy sharedsymped-b after the conversion. Each row in the table represents a different number of writer tasks from 200 to 800. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for N-copy sharedsymped-b with readers/writer locks is around 700 processes and 60,000 connections (15.88), with a peak throughput of 5416 Mbps occurring at 50,000 requests per second and with a sustained throughput of around 4614 Mbps at 70,000 requests per second. The throughput is 5% higher at peak than the throughput of N-copy sharedsymped-b and 4% higher at 70,000 requests per second.

Based on vmstat output, there is no idle time at peak and the idle time at 70,000 requests per second is 5%. Unfortunately, by 70,000 requests per second the system starts to experience a small amount of disk I/O. Further increasing the number of processes to 800 improves the throughput of the server at 70,000 requests per second and eliminates idle time at the cost of decreased throughput at peak. Increasing both the number of processes to 800 and the maximum connections value to 80,000 results in throughput gains at 70,000 requests per second with a smaller decrease in peak throughput. Unfortunately, the memory footprint of the server becomes too large so the overall effectiveness of further increasing the number of processes is muted as disk I/O starts to occur. In terms of overall performance, the trade off in peak throughput versus sustained throughput for these experiments did not result in a higher condensed area.

4.6.1.2 Tuning N-copy WatPipe

Extensive changes were made to WatPipe to improve its multiprocessor performance. These changes are described in Section 4.6.3. This new version of WatPipe is used for the N-copy experiments in this section.

Experiments were run to tune the two versions of N-copy WatPipe: N-copy watpipe-nb and N-copy watpipe-b. For all versions of WatPipe, the parameters tuned are the maximum number of simultaneous connections and the number of writer tasks.

Similar to the *μ*server experiments, one copy of WatPipe is started on each processor, resulting in 4 copies of WatPipe for these experiments. In the tables, the number of writers and maximum number of connections listed represents the aggregate across all copies of the server. The values for each individual server can be obtained by dividing by four as writer tasks and connections are equally distributed among the servers.

Table 4.8(a) shows the results of tuning N-copy watpipe-nb. Each row in the table represents a different number of writer tasks from 4 to 16. The columns represent a different maximum number of connections from 40,000 to 100,000. The rows and columns represent the total number of writer tasks and connections across all the servers. The experiments show the best performance for N-copy watpipe-nb is around 12 writer tasks and 100,000 connections (16.92), with a peak throughput of 6066 Mbps occurring at 56,000 requests per second and with a sustained throughput of around 5705 Mbps at 70,000 requests per second. Unexpectedly, the best performance does not occur with 4 writer tasks (one task per CPU). However, the highest peak does occur with 4 writer tasks and 80,000 connections. This peak is 6196 Mbps at 56,000 requests per second. This peak is very close to the peak of 6202 Mbps achieved by the N-copy symped-nb server.

The experiment with 12 writer tasks has a larger condensed area because its post-peak performance experiences a more gradual decline in throughput. While the best performing N-copy symped-nb and N-copy watpipe-nb with 4 writer tasks experience a 6.5% and 8.1% decline in throughput at 58,000 request per second respectively, the 12 writer version of N-copy watpipe-nb experiences a decline of only 1.1%. While the sustained throughput for all three servers is around 5700 Mbps, the more gradual decline in performance of N-copy watpipe-nb with 12 writers results in a higher condensed area despite having a lower peak. This result is unexpected and interesting. Similarly, the shaper decline in throughput for the 4 writer version of N-copy watpipe-nb resulted in a lower condensed area for N-copy watpipe-nb with 80,000 connections.

Table 4.8(b) shows the results of tuning of N-copy watpipe-b. Each row in the table represents a different number of writer tasks from 200 to 600. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for N-copy watpipe-b is

Writers	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.62	16.81	**16.81**	16.80
8	12.60	16.80	16.72	16.73
12	12.61	16.76	16.91	**16.92**
16	12.61	16.74	16.87	16.87

(a) non-blocking sendfile

Writers	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	12.59	15.18	15.11	14.75
300	12.61	16.54	16.44	16.15
400	12.60	16.71	**16.79**	16.78
500	12.60	16.68	16.73	16.73
600	12.60	16.64	16.68	16.68

(b) blocking sendfile

Table 4.8: N-copy WatPipe experiments - 4 GB

around 400 writer tasks and 80,000 connections (16.79), with a peak throughput of 5957 Mbps occurring at 56,000 requests per second and with a sustained throughput of around 5645 Mbps at 70,000 requests per second. The difference in peak throughput between the best N-copy non-blocking server (N-copy symped-nb) and N-copy watpipe-b is around 4%, which is in line with the 4% throughput difference observed between the best non-blocking and blocking servers in the 4 GB uniprocessor experiments. The throughput is 10% higher at peak than the throughput of N-copy sharedsymped-b with readers/writer locks and 22% higher at 70,000 requests per second. The performance of N-copy watpipe-b is more consistent with the performance of the blocking servers in Chapter 3 than the other multiprocessor N-copy blocking-servers.

One reason for the stability of the N-copy watpipe-b server is that its memory footprint is smaller than the memory footprint of the other N-copy *μ*server servers. Hence, the entire file set fits into the file-system cache for all the N-copy watpipe-b experiments. However, even for experiments with the same parameters where neither server experiences disk I/O, N-copy watpipe-b performs better. N-copy watpipe-b has its best performance at 400 writer tasks and 80,000 connections and N-copy sharedsymped-b with readers/writer locks also has no disk I/O for those parameters; however, N-copy sharedsymped-b's performance is worse than N-copy watpipe-b. In fact, N-copy sharedsymped-b with readers/writer locks peaks at 50,000 requests per second with a throughput of 5460 Mbps but only has a throughput of 5295 Mbps at 56,000 requests per second. N-copy watpipe-b has equivalent performance at 50,000 requests per second, but peaks at 56,000 requests per second with a throughput of 5957 Mbps. One major difference among the servers is that with the shared-SYMPED server, each process is a separate event-driven server. For 400 processes and 80,000 connections at 56,000 requests per second, N-copy sharedsymped-b with readers/writer locks has an average of 8592 calls to epoll_wait per second while N-copy watpipe-B has 2007 calls per second on average. Based on OProfile data for this experiment, N-copy watpipe-b spends 1.79% of time in the kernel event-mechanism versus 3.29% for N-copy sharedsymped-b with readers/writer locks. This difference accounts for some but not all of the performance difference.

Not only do N-copy sharedsymped-b and N-copy watpipe-b achieve their best performance with different tuning parameters, but the performance of N-copy watpipe-b is stable over a larger tuning range than N-copy sharedsymped-b. Switching to a readers/writer lock helps to increase the performance and scalability of N-copy sharedsymped-b but a large number of processes are required to achieve best performance. One big difference between N-copy watpipe-b and N-copy sharedsymped-b is the contention on the locks associated with the cache table. With N-copy watpipe-b, only reader tasks contend for the readers/writer lock and with the N-copy version there is only one reader task per server so there is no contention on the lock. The reader task and writer tasks only contend for the individual locks associated with each cache entry. With N-copy sharedsymped-b, all the processes associated with a server copy contend for both the readers/writer lock associated with the cache table and the locks associated with individual cache entries. Despite the fact that all the processes sharing the lock execute on the same CPU, the contention on the lock reduces concurrency and increases overhead because processes tend to be blocked waiting for the lock. Overcoming the reduction in concurrency requires more processes, which increases execution overhead and the memory footprint of the server, resulting in lower throughput. It is interesting to note that N-copy sharedsymped-nb avoids this problem by reducing contention as it only requires a small number of processes. However, this solution is not viable for the blocking server.

The expectations for these experiments is that N-copy should produce the best performance [62]. Requests are independent so the server processes can be independent. However, non-N-copy servers offer advantages that can be useful, such as shared memory, better load balancing, etc. The next sections examine these types of servers and what can be done to offer reasonable performance.

4.6.2 Tuning μserver

Experiments were run to tune three versions of μserver: symped-nb, sharedsymped-nb and sharedsymped-b. Symped-b is excluded because its memory footprint is too large. For all versions of μserver, the parameters tuned are the maximum number of simultaneous connections and the number of processes.

Table 4.9 shows the results of tuning symped-nb. Each row in the table represents a different number of processes from 4 to 16. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for symped-nb is around 4 processes and 80,000 connections (16.13), with a peak throughput of 5463 Mbps occurring at 50,000 requests per second and with a sustained throughput of around 5217 Mbps at 70,000 requests per second. Experiments with additional rates between 50,000 and 56,000 requests per second were run to explore the area of peak throughput for the server (not shown). Based on these experiments, the peak throughput for symped-nb with 4 processes and 80,000 connections is 5585 Mbps occurring at 52,000 requests per second. N-copy

	Maximum Number of Connections			
Procs	40,000	60,000	80,000	100,000
4	12.81	16.12	**16.13**	16.11
8	12.82	16.06	16.03	16.04
12	12.81	16.07	15.99	15.99
16	12.81	16.08	15.96	15.95

Table 4.9: μserver non-blocking SYMPED experiments - 4 GB

symped-nb has throughput that is approximately 11% higher at peak than symped-nb and 9% higher at 70,000 requests per second

There are a couple of differences between this experiment and the N-copy version. First, none of the processes have CPU affinity set, so any process is free to execute on any CPU. Furthermore, only a single listening port is used so each server process can handle connections from any subnet. Note, interrupt processing is still equally distributed so that two network interfaces are pinned to each CPU.

These differences result in a server that is distinct from the N-copy servers examined in Section 4.4. Specifically, the performance of symped-nb is lower than the performance of the N-copy symped-nb server without process affinities, even though both run with network-interface interrupt-affinities set on the system but not process affinities. In the N-copy symped-nb server without process affinities, each process only handles requests from two subnets, both associated with a single CPU; it is possible for the operating system to schedule a process on the CPU associated with its subnets. In fact, the Linux scheduler does a reasonable job of scheduling the processes of the N-copy server and the server achieves good performance (see Section 4.4). With symped-nb, however, a process can handle connections from any subnet regardless of the CPU on which it is executing. Since the affinity of the network interfaces associated with these subnets span multiple CPUs, scheduling the server processes in order to maintain CPU affinity between processes, subnets and requests is impossible without more support from the operating system. The inability to partition symped-nb results in increased overhead and lower performance than N-copy symped-nb. For example, with 4 processes and 80,000 connections at 50,000 requests per second, average softirq time is 8% higher and average system time is 7% higher for symped-nb than both N-copy symped-nb and N-copy symped-nb server without process affinities. The increased overhead means that the symped-nb server peaks earlier with lower throughput.

However, symped-nb performs better than the N-copy symped-nb server with misaligned network interface and process affinities. With the misaligned experiment, each request is handled by a process executing on a CPU that is different from the CPU tied to the subnet of the request. With symped-nb, the expectation is that 25% of the requests are handled by processes executing on the same CPU as the network interface associated with the subnet of the request, resulting in better performance than the misaligned server with 0%.

In order to improve the performance of symped-nb, the logical step is to restrict the server processes so they do not handle requests from all subnets. Partitioning the subnets among the server processes would allow these processes to be scheduled on the appropriate CPU. This partitioning is essentially the N-copy server discussed earlier in the chapter.

Table 4.10(a) shows the results of tuning sharedsymped-nb. Each row in the table represents a different number of processes from 4 to 16. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for sharedsymped-nb is around 4 processes and 60,000 connections (15.97), with a peak throughput of 5462 Mbps occurring at 50,000 requests per second and with a sustained throughput of around 5097 Mbps at 70,000 requests per second. N-copy symped-nb has approximately 14% higher throughput at peak than sharedsymped-nb and 12% higher at 70,000 requests per second. As well, symped-nb has slightly higher throughput at peak and a more gradual decline in throughput after peak, resulting in better overall performance than sharedsymped-nb.

Based on vmstat output gathered during the experiment, sharedsymped-nb has about 2% idle time even after peak. At the application level, only the cache table is shared among the server processes. Moving to better locking on the shared cache-table, such as a readers/writer lock, should improve scalability and eliminate idle time. Table 4.11(a) shows the results of tuning sharedsymped-nb after converting the shared cache-table to use readers/writer locks. This change resulted in a small performance improvement as the idle time now goes down to zero. The experiments show the best performance for sharedsymped-nb with readers/writer locks is around 4 processes and 80,000 connections (16.16). The peak throughput is still 5463 Mbps but sustained throughput is around 5266 Mbps at 70,000 requests per second. Not only is this performance consistent with the results for symped-nb, but both servers spend a larger amount of time servicing softirqs compared to N-copy symped-nb. With 4 processes and 80,000 connections at 50,000 requests per second, the increase in average time spent on softirqs is about 3% and the increase in average system time is about 8% compared to N-copy symped-nb. This overhead is less than the overhead experienced by the symped-nb server. While the sharedsymped-nb server with readers/writer locks has a slightly lower peak than the symped-nb server, it has better throughput at 70,000 requests per second.

The sharedsymped-nb servers have two disadvantages over their N-copy counterparts. First, the cache table is shared across processors, inhibiting parallelism. This problem is mitigated by switching to readers/writer locks for the cache table. The second problem is the inability to partition server processes, subnets and CPUs. In order to address the second problem, a version of sharedsymped-nb is implemented such that processes, subnets and CPUs are partitioned but a single cache table is shared across all processes. The difference between N-copy symped-nb and sharedsymped-nb with readers/writer locks and

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.82	**15.97**	15.97	15.94
8	12.81	15.35	15.35	15.38
12	12.82	14.89	14.88	14.89
16	12.81	14.74	14.73	14.73

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	11.67	12.89	11.77	11.02
300	12.05	**13.53**	13.39	12.84
400	11.87	13.19	13.25	13.20
500	11.65	12.84	12.82	12.80
600	11.41	12.48	12.42	12.26

(b) blocking sendfile

Table 4.10: *μ*server shared-SYMPED experiments - 4 GB

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.82	16.13	**16.16**	16.15
8	12.81	16.07	16.01	16.00
12	12.81	16.06	15.96	15.96
16	12.81	16.06	15.93	15.93

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	11.69	13.40	12.64	12.06
300	12.08	**13.87**	13.64	13.01
400	11.92	13.54	13.59	13.49
500	11.69	13.15	13.19	13.08
600	11.44	12.68	12.73	12.66

(b) blocking sendfile

Table 4.11: *μ*server shared-SYMPED with readers/writer locks experiments - 4 GB

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.87	**16.84**	16.80	16.81
8	×	×	16.42	16.44
12	12.88	×	16.33	16.32
16	12.87	×	16.29	16.30

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	12.29	14.41	13.35	×
300	12.66	15.44	14.65	13.88
400	12.74	15.71	15.41	14.83
500	12.83	15.71	15.52	15.21
600	12.95	**15.82**	15.64	15.44
700	13.04	15.70	15.67	×

(b) blocking sendfile

Table 4.12: *μ*server shared-SYMPED with readers/writer locks and process affinities experiments - 4 GB

process affinities is that sharedsymped-nb has a shared cache-table and its processes share file descriptors. The shared cache-table affects performance, but its effect is small when the number of processes is also small.

Table 4.12(a) shows the results of tuning sharedsymped-nb with readers/writer locks and process affinities. Each row in the table represents a different number of processes from 4 to 16. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for sharedsymped-nb with readers/writer locks and process affinities is around 4 processes and 60,000 connections (16.84), with a peak throughput of 5959 Mbps occurring at 56,000 requests per second and with a sustained throughput of around 5673 Mbps at 70,000 requests per second. The throughput of sharedsymped-nb with readers/writer locks and process affinities is around 4% lower at peak than N-copy symped-nb and approximately the same at 70,000 requests per second.

At peak, the server experiences an average of 6% idle time that reduces to 1% by 70,000 requests per second. Increasing the maximum number of connections reduces the idle time but does not improve performance.

Table 4.10(b) shows the results of tuning sharedsymped-b. Each row in the table represents a different number of processes from 200 to 600. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for sharedsymped-b is around 300 processes and 60,000 connections (13.53), with a peak throughput of 4349 Mbps occurring at 45,000 requests per second and with a sustained throughput of around 3999 Mbps at 70,000 requests per second. The difference in peak throughput between N-copy sharedsymped-b and sharedsymped-b is around 18% and N-copy sharedsymped-b also has 11% higher throughput at 70,000 requests per second. This throughput difference is larger than with the non-blocking version.

There are two major differences between the sharedsymped-b and N-copy sharedsymped-b servers. First, N-copy sharedsymped-b benefits from additional parallelism as there are four separate cache tables, one per CPU; whereas, sharedsymped-b has a single cache table that is shared by all the server processes. Furthermore, with N-copy sharedsymped-b, only processes tied to the same CPU share a cache table. Hence, the shared cache-table does not inhibit the potential parallelism across the system as blocking effects are limited to a single CPU and the amount of contention on each cache table is less since only 25% of the processes share a single cache table. Second, N-copy sharedsymped-b benefits from the alignment of CPUs, network interfaces and processes, while the processes in sharedsymped-b accept connections from any subnet and are not tied to a single CPU. The expectation is that 25% of requests should be aligned while the remainder of the requests should not be aligned. As shown previously, this alignment can make a large difference in performance.

Since using a single cache table is a bottleneck that inhibits the server from scaling, a new version of *μ*server is implemented with a readers/writer lock used for the shared cache-table. Table 4.11(b) shows the results of tuning sharedsymped-b with readers/writer locks used for the shared cache-table. The experiments show the best performance for sharedsymped-b with readers/writer locks is around 300 processes and 60,000 connections (13.87), with a peak throughput of 4528 Mbps occurring at 45,000 requests per second and with a sustained throughput of around 3972 Mbps at 70,000 requests per second. Compared to N-copy sharedsymped-b, the throughput of sharedsymped-b with readers/writer locks is around 12% lower at peak and 11% lower at 70,000 requests per second.

The effect of switching to readers/writer locks is small for the blocking server; however, that is expected as the experiments have no idle time. With 300 processes and 60,000 connections at 45,000 requests per second, sharedsymped-b with readers/writer locks spends approximately 66% of its time handling softirqs, 12% more than N-copy sharedsymped-b with readers/writer locks. It is interesting to note that sharedsymped-b with readers/writer locks actually has lower system and user times than the N-copy version. This difference can be attributed to the lower throughput of sharedsymped-b with readers/writer locks. Since, handling softirqs takes more CPU time, the experiment peaks earlier with lower throughput. Not partitioning processes, subnets and CPUs has a detrimental effect on performance and the additional processes required by sharedsymped-b only exacerbates the situation.

Table 4.12(b) shows the results of tuning sharedsymped-b with readers/writer locks and process affinities. Each row in the table represents a different number of processes from 200 to 700. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for sharedsymped-b with readers/writer locks and process affinities is around 600 processes and 60,000 connections (15.82), with a peak throughput of 5452 Mbps occurring at 50,000 requests per second and with a sustained throughput of around 4568 Mbps at 70,000 requests per second. The difference in throughput between N-copy sharedsymped-b with readers/writer locks and sharedsymped-b with readers/writer locks and process affinities is less than 1% at both peak and 70,000 requests per second.

At peak, the server experiences no idle time but this value increases to 7% by 70,000 requests per second. As expected, since sharedsymped-b with readers/writer locks and process affinities uses a single cache table across all processes and CPUs, its overall performance is a little lower. However, both servers start to experience disk I/O, which has a tendency to even out the performance. It turns out that sharedsymped-b with readers/writer locks and process affinities has a larger memory footprint than its N-copy counterpart. The memory savings gained by only having one cache table instead of four is more than offset by the increase in the size of some application data structures resulting from sharing file descriptors. While the number of file descriptors remains constant for both servers, the maximum file-descriptor supported by an individual process is four times larger with sharedsymped-b compared to N-copy sharedsymped-b. In particular, μserver uses an index array to map a socket descriptor to its corresponding entry in an array containing the information about its associated request. Each entry in the index array consists of a single integer and the number of elements in the array is based on the largest possible file descriptor for the server. For N-copy sharedsymped-b, with 600 processes and 60,000 connections the maximum file descriptor is approximately 40,000 (15,000 connections per N-copy $+$ 25,000 files in the file-set), requiring 92 MB of memory (600 processes $\times$ 4 bytes $\times$ 40,000 entries). For sharedsymped-b with similar parameters, the maximum file descriptor is approximately 85,000 (60,000 connections $+$ 25,000 files), requiring 195 MB of memory (600 processes $\times$ 4 bytes $\times$ 85,000 entries). However, the cache table is only 6 MB, so four cache tables require 24 MB of memory. While sharedsymped-b saves 18 MB of memory by only having one cache, it expends 103 MB of memory due to larger index arrays. This increase in memory footprint causes sharedsymped-b with readers/writer locks and process affinities to peak with fewer processes as it incurs disk I/O with fewer processes, which begins to occur at 600 processes and 60,000 connections.

With SYMPED and shared-SYMPED, a single process handles an entire request so the difference between these servers and their N-copy counterparts is small. However, based on the results for the SYMPED and shared-SYMPED servers in this section, aligning requests, processes, subnets and CPUs is important for achieving the best performance. Adopting this partitioning to create a hybrid server improved performance but the difference between the hybrid shared-SYMPED server and N-copy is small. Other servers, for example, WatPipe, are more amenable to hybrid approaches that are still somewhat distinct from N-copy.

4.6.3 Tuning WatPipe

Major changes are required for WatPipe to run well with these experiments. The goal of changing WatPipe is to create a hybrid server that encompasses the major benefits of an N-copy server while retaining advantages of a shared-memory pipeline server. The most important feature of the N-copy approach is the ability to partition server processes, subnets and CPUs in order to reduce overhead and improve throughput. The challenge with WatPipe is to determine which parts of the pipeline require partitioning.

Completely partitioning the kernel threads and data structures would essentially result in an N-copy server in a single address space. While certain types of servers may benefit from this approach as it allows data or computations to be efficiently shared via common memory, it offers no benefits over straight N-copy for the static workload experiments in this section. In fact, the need to support a large number of file descriptors due to the shared address-space is a drawback compared to actually running N-copy. Overall, the biggest problem with the N-copy approach is the large memory footprint of the server due to duplication resulting from independent data structures. The advantages of this approach are reduced overhead due to partitioning and reduced contention as server data is not shared across the partitioned kernel threads.

The other extreme is for the server to be completely unpartitioned. In this case, the tasks in the server are free to handle requests from any subnet and can execute on any CPU. Based on the experiments in the previous section, there is a significant penalty associated with not partitioning the server. As well, the shared queues and other data structures represent a major source of contention. Locking the queues inhibits parallelism across the CPUs and represents a significant bottleneck. The major benefit of this approach is that the memory footprint of the server is small compared to the N-copy approach. Additional benefits include centralized event polling and CPU load balancing since tasks can execute on any CPU.

A hybrid approach is to selectively partition sections of the pipeline. Specifically, only sections of the pipeline directly involved with network I/O are partitioned. In the case of WatPipe, the reader and writer tasks and their associated queues are partitioned and each subnet is allocated an equal portion of the maximum number of connections. Advantages of moving to a hybrid approach include partitioning to reduce overhead, efficiencies due to centralizing event polling and a small amount of load balancing is possible since the remaining tasks are free to execute on any CPU. There are a number of tradeoffs resulting from moving to a hybrid approach. While the memory footprint is smaller than N-copy, it is not as small as the unpartitioned approach. Contention due to shared queues and data structures is higher than the N-copy approach but reduced compared to the unpartitioned approach since queues are only selectively shared.

While completely partitioning the server likely yields the best performance, it offers no benefits compared to N-copy for these experiments while possibly incurring additional overhead. On the other hand, the performance of a completely unpartitioned server is too low for it to be considered a reasonable choice. Hence, the hybrid approach is chosen for WatPipe as it offers the best compromise, while incurring only a small performance penalty compared to N-copy.

Specifically, the hybrid approach involves partitioning selective sections of the pipeline. The idea behind partitioning is to associate a task with a CPU and for that task to only handle requests from subnets tied to the same CPU. Since there are four CPUs with two associated subnets, where applicable, pipeline stages are similarly partitioned. Hence, reader and writer tasks are assigned exclusively to a particular CPU with each type of task equally distributed among the CPUs. Therefore, each reader or writer task only executes on a single CPU and services the two subnets associated with that CPU. To reduce contention, there is a separate queue per CPU for the Read and Write stages of the pipeline. For these experiments, exactly four reader tasks are used, one assigned to each CPU. The number of writer tasks is one of the parameters that is varied during tuning to achieve the best performance. Since the maximum number of connections is equally divided between the subnets, a single listening socket does not provide enough control because an accepted connection could come from any of the subnets. Therefore, a separate listening socket is created for each subnet with one acceptor task per subnet, resulting in eight acceptor tasks. While each acceptor task is associated with a specific subnet, they are free to execute on any CPU as no affinities are set for these tasks. There is still only one task handling polling for read events for all subnets and one task handling polling for write events for all subnets. Since both of these tasks are free to execute on any CPU, their execution is throttled by introducing a small delay between polls.

While this approach seems very similar to N-copy, it has some significant differences. Aside from the

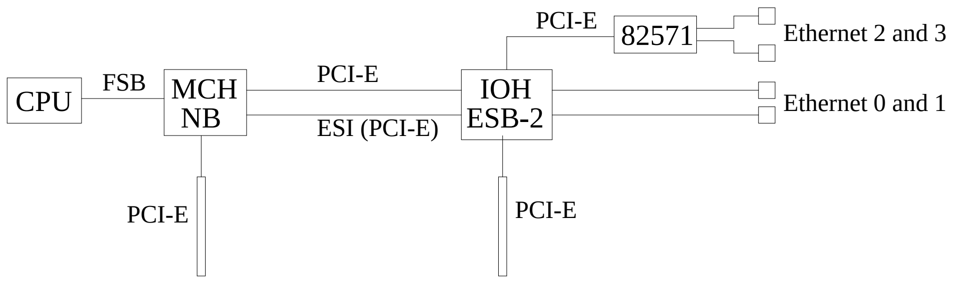


Figure 4.2: Partial block diagram of server hardware

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 cache-table. Similar to the changes made to the μserver cache table, readers/writer locks are used to allow concurrent access to the cache table. Hence, the hybrid implementation has a smaller memory footprint than the N-copy version. Not all tasks are confined to a single CPU and some activities are handled by a single task instead of having a separate task per subnet or CPU. More specifically, the acceptor, ReadPoll and WritePoll tasks do not have affinities set and can execute on any processor. The advantage of allowing these tasks to float is that if offers some flexibility for the scheduler to perform a small amount of load balancing to better handle small variations in load.

Experiments were run to tune two versions of WatPipe: watpipe-nb and watpipe-b. For all versions of WatPipe, the parameters tuned are the maximum number of simultaneous connections and the number of writer tasks.

Table 4.13(a) shows the results of tuning watpipe-nb. Each row in the table represents a different number of writers from 4 to 16. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for watpipe-nb is around 4 writer tasks and 100,000 connections (17.04), with a peak throughput of 6070 Mbps occurring at 56,000 requests per second and with a sustained throughput of around 5835 Mbps at 70,000 requests per second. The difference in peak throughput between N-copy watpipe-nb and watpipe-nb is around 2% and between N-copy symped-nb and watpipe-nb is also around 2%, with watpipe-nb having lower peak throughput in both cases. At 70,000 requests per second, watpipe-nb's throughput is 2% better than N-copy watpipe-nb and 2% better than N-copy symped-nb. But just after peak at 60,000 requests per second, watpipe-nb's throughput is 6% higher than N-copy watpipe-nb and 4% higher than N-copy symped-nb. This stability is even more impressive given that the peak of watpipe-nb is only 2% lower than the best N-copy servers. A more gradual decline after peak gives watpipe-nb a larger condensed area than the N-copy servers despite having a lower peak.

Writers	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.61	16.78	17.03	**17.04**
8	12.60	16.77	16.93	16.92
12	12.59	16.45	16.48	16.47
16	12.60	16.43	16.45	16.45

(a) non-blocking sendfile

Writers	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
200	12.58	15.02	14.89	14.37
300	12.60	16.37	16.29	16.11
400	12.61	16.39	**16.41**	16.40
500	12.59	16.34	16.36	16.36
600	12.60	16.29	16.30	16.29

(b) blocking sendfile

Table 4.13: WatPipe experiments - 4 GB

Procs	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.86	16.84	17.09	**17.10**

Table 4.14: μserver N-copy non-blocking SYMPED load balancing experiments - 4 GB

Writers	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.61	16.82	**17.03**	17.02

Table 4.15: N-copy non-blocking WatPipe load balancing experiments - 4 GB

Writers	Maximum Number of Connections			
	40,000	60,000	80,000	100,000
4	12.60	16.80	**17.28**	17.27

Table 4.16: Non-blocking WatPipe load balancing experiments - 4 GB

Figure 4.2 contains a portion of the system-level block diagram for the server machine used in the experiments [51]. Specifically, the diagram shows the connection of the on-board network interfaces and the PCI slots containing the additional Ethernet ports. Ethernet 0 and 1 are connected directly to the Southbridge I/O Hub (IOH), Ethernet 2 and 3 are connected via a dual gigabit Ethernet chip (Intel Ophir 82571) to the IOH via a PCI-E connection and the remaining Ethernet interfaces are add-on cards connected via PCI-E to the Northbridge Memory Controller Hub (MCH) and the IOH respectively. The network interfaces, not directly connected to the IOH have additional latencies and CPU overhead, because they must communicate over the PCI bus, compared to the network interfaces connected directly to the

IOH. These lower-overhead network-interfaces were not used for the main experiments in this chapter, however, several experiments were run using these network interfaces to see the effect of a small hardware performance-imbalance on the servers. As a result of partitioning, two of the CPUs on the machine are associated with one lower-overhead network-interface and one higher-overhead network-interface, while the other two CPUs are associated with two higher-overhead network-interfaces. Table 4.14 shows the results of tuning N-copy symped-nb, Table 4.15 shows the results of tuning N-copy watpipe-nb and Table 4.16 shows the results of tuning watpipe-nb. All the servers were run with 4 processes or writers across all copies of the server and with a maximum number of connections from 40,000 to 100,000. All the servers show a performance improvement due to the lower-overhead direct-connection. N-copy symped-nb has best performance at 100,000 connections (17.10), with a throughput of 6202 Mbps at peak and 5897 Mbps at 70,000 requests per second. N-copy watpipe-nb has best performance at 80,000 connections (17.03), with a throughput of 6196 Mbps at peak and 5907 Mbps at 70,000 requests per second. Watpipe-nb has best performance at 80,000 connections (17.28), with a throughput of 6293 Mbps at peak and 6051 Mbps at 70,000 requests per second.

Because of the imbalance, the two CPUs only associated with the higher-overhead network-interfaces become saturated before the other two CPUs. The N-copy servers show a small increase in overall performance but not in peak throughput. With the N-copy servers, the higher-overhead network-interfaces cannot handle an increase in capacity, which throttles the throughput of the entire system. Watpipe-nb, however, has higher throughput at peak and higher overall performance. The reason watpipe-nb achieves higher throughput is because not all of its tasks are confined to a specific CPU. Therefore, the free tasks can be scheduled on the less saturated CPUs, freeing up processing capacity on the CPUs only associated with the higher-overhead network-interfaces. Examining the idle time of the servers at 58,000 requests per second, the N-copy servers have an average of 4–5% idle time on two of the CPUs and 10–13% on the other two CPUs, but watpipe-nb has an average of only 3–5% idle time on all four CPUs. The end result is watpipe-nb can take advantage of a small amount of load balancing to achieve higher throughput than the N-copy servers.

As mentioned, the lower-overhead network-interfaces were only used for the previous few experiments. Therefore, these experiments are not discussed further, for example, in the Server Comparison section. The remaining experiments in this chapter all use the same hardware configuration that does not include these alternative network interfaces.

Table 4.13(b) shows the results of tuning watpipe-b. Each row in the table represents a different number of writers from 200 to 600. The columns represent a different maximum number of connections from 40,000 to 100,000. The experiments show the best performance for watpipe-b is around 400 writer tasks and 80,000 connections (16.41), with a peak throughput of 5622 Mbps occurring at 56,000 requests per second and with a sustained throughput of around 5395 Mbps at 70,000 requests per second. Experiments with additional rates between 50,000 and 56,000 requests per second were run to explore the area of peak throughput for the server (not shown). Based on these experiments, the peak throughput for watpipe-b with 400 writer tasks and 80,000 connections is 5729 Mbps occurring at 54,000 requests per second. The throughput of N-copy watpipe-b is approximately 4% higher at peak than watpipe-b and 5% higher at 70,000 requests per second.

N-copy watpipe-b and watpipe-b have their best performance with the same configuration parameters. However, based on vmstat data, watpipe-b has an average of 4% idle time at peak and 3% at 70,000 requests per second, while N-copy watpipe-b has an average of 2% idle time at peak and no idle time at 70,000 requests per second. These differences occur because N-copy watpipe-b has less contention than watpipe-b and no locking or sharing of application data across CPUs.

Watpipe-b has 5% better throughput than sharedsymped-b and 6% better throughput than N-copy sharedsymped-b at peak. Both sharedsymped-b and N-copy sharedsymped-b have their best performance with more processes than the number of writer tasks in watpipe-b: 700 processes for N-copy sharedsymped-b and 600 for sharedsymped-b. An interesting difference between watpipe-b and sharedsymped-b involves the amount of contention on the cache-table readers/writer lock. With sharedsymped-b, all the processes are symmetric and must acquire the lock on the cache-table at some point while processing a request. Even though most of the acquisitions are for reading, all lock calls require some basic mutual exclusion to determine the lock state, and also exclusive access is required on the first request for a file. Hence, there are hundreds of processes across multiple CPUs contending for the cache-table lock. With watpipe-b, only the reader tasks acquire the cache-table lock as the writer tasks work with the mutex lock associated with individual cache entries. Hence, with watpipe-b there are only four tasks contending for the cache-table lock. As well, since there are dedicated tasks servicing the various stages of the pipeline, when a reader task blocks on the cache-table lock, the other tasks in the system can continue execution. In order to compensate for blocking on the cache-table lock, sharedsymped-b requires additional processes, increasing overhead. Eventually, the memory footprint of sharedsymped-b becomes large enough that disk I/O occurs as the file-system cache can no longer hold the entire file set causing the throughput of the server to be capped. While other factors may influence the performance of the servers, the effect of disk I/O is too large to overcome. The fact that watpipe-b has a small memory footprint also accounts for its stability as it does not have disk I/O for any of the tuning parameters tested.

4.6.4 Server Comparison

Figure 4.3 presents the best performing configuration for each server-architecture implementation: μserver N-copy non-blocking SYMPED, μserver N-copy blocking shared-SYMPED with readers/writer locks, N-copy non-blocking WatPipe, N-copy blocking WatPipe, μserver non-blocking SYMPED, μserver

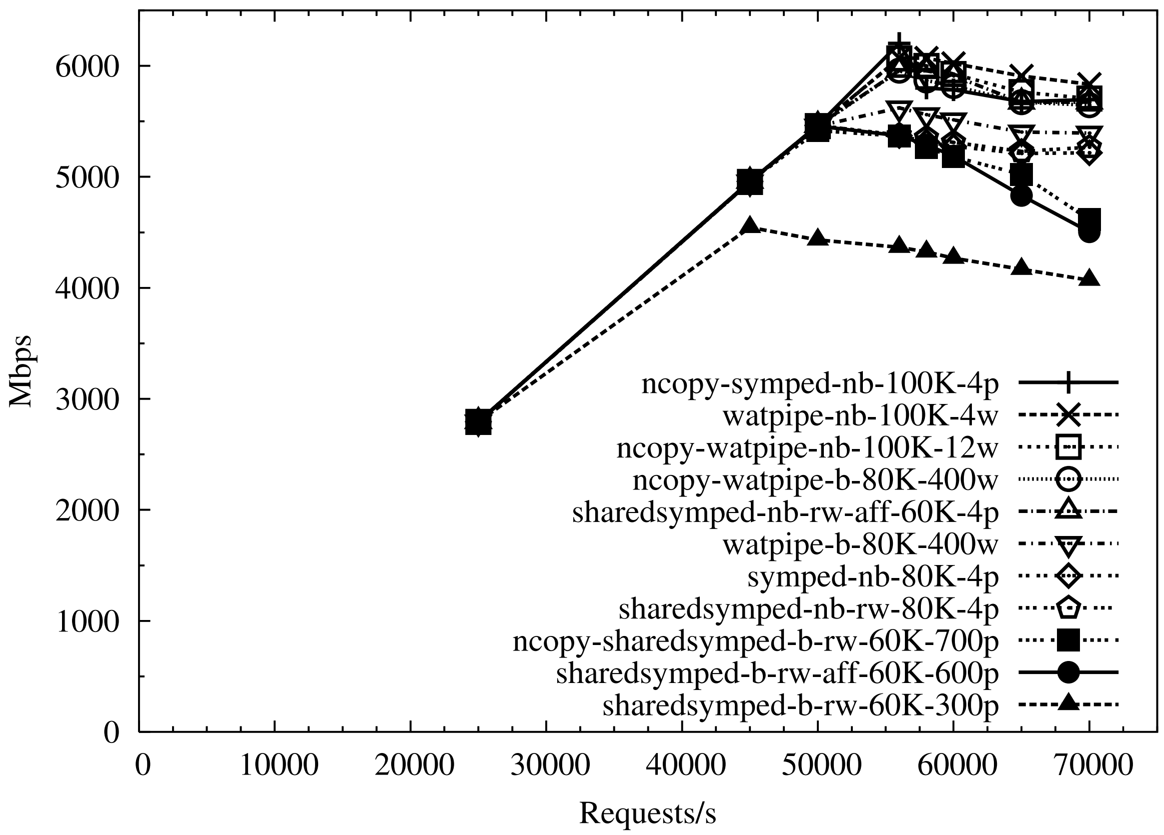


Figure 4.3: Throughput of different architectures - 4 GB

Server	Rank
watpipe-nb	1
N-copy symped-nb	2
N-copy watpipe-nb	2
N-copy watpipe-b	3
sharedsymped-nb with readers/writer locks and process affinities	3
watpipe-b	4
N-copy sharedsymped-b with readers/writer locks	5
sharedsymped-b with readers/writer locks and process affinities	5

Table 4.17: Ranking of server performance - 4 GB

non-blocking shared-SYMPED with readers/writer locks, *μ*server blocking shared-SYMPED with readers/writer locks, *μ*server non-blocking shared-SYMPED with readers/writer locks and process affinities, *μ*server blocking shared-SYMPED with readers/writer locks and process affinities, non-blocking WatPipe and blocking WatPipe. The legend in Figure 4.3 is ordered from the best performing server at the top to the worst at the bottom. In the legend, "rw" indicates the use of readers/writer locks and "aff" indicates process affinities for the shared-SYMPED servers. Excluding sharedsymped-b with readers/writer locks, peak server throughput varies by about 15% (5416–6202 Mbps), a range of 786 Mbps.

Table 4.17 ranks the performance of the servers for the 4 GB workload. Only the best server of each type is included in the ranking, so all the non-N-copy servers without process affinities are excluded from the ranking. Again, based on a total of three runs for each server, Tukey's Honest Significant Difference test is used to differentiate the servers with a 95% confidence level. The servers are then ranked based on mean area.

The top performer is watpipe-nb, followed by N-copy symped-nb and N-copy watpipe-nb, which have approximately the same performance. The next grouping consists of N-copy watpipe-b and sharedsymped-nb with readers/writer locks and process affinities. Watpipe-b is next followed by the blocking shared-SYMPED servers, N-copy sharedsymped-b with readers/writer locks and sharedsymped-b with readers/writer locks and process affinities. The non-blocking servers without process affinities (not ranked), symped-nb and sharedsymped-nb with readers/writer locks, appear to have performance between watpipe-b and the blocking shared-SYMPED server. The blocking shared-SYMPED servers, have approximately the same peak as the non-blocking servers without process affinities but larger decreases in throughput after peak. Sharedsymped-b with readers/writer locks (not ranked) is last with the worst performance. Comparing the performance of the best version of WatPipe and the best version of *μ*server, N-copy symped-nb has a 2% higher peak at 56,000 requests per second but watpipe-nb has 2% higher throughput at 70,000 requests per second. If only peak throughput is considered, then N-copy watpipe-nb with its best peak (not shown in Figure 4.3 as it has a lower condensed area than the corresponding server in the figure) and N-copy symped-nb have approximately the same throughput at peak and at 70,000 requests per second. Ignoring the N-copy servers and comparing the best version of WatPipe and the best version of *μ*server shared-SYMPED, watpipe-nb has 2% higher throughput than sharedsymped-nb with readers/writer locks and process affinities. For the non-blocking servers, both WatPipe and the best *μ*server servers have similar performance. The real difference occurs among the blocking servers, with WatPipe having a clear advantage. The best blocking WatPipe server, N-copy watpipe-b, has 9% higher peak throughput than the best blocking *μ*server, sharedsymped-b with readers/writer locks and process affinities.

Finally, consider the difference between the N-copy and single-copy versions of a server. For non-blocking WatPipe, the differences in peak throughput is approximately 2% and for blocking WatPipe around 4%. For non-blocking shared-SYMPED, the difference in peak is 4% (assuming the same peak as for N-copy symped-nb), and for blocking shared-SYMPED less than 1%. For these experiments, the performance advantage of N-copy over the hybrid servers is small.

To better understand the performance of the servers, the best configuration of each server is profiled. The OProfile, vmstat and mpstat data for these experiments are summarized in Tables 4.18 and 4.19. Some additional terms are used to describe the architecture of the multiprocessor servers. In this section, “N-copy” means the server is being run N-copy with each server copy and its associated subnets partitioned to execute on a separate CPU, “rw” means that *µ*server is being run with a readers/writer lock for its cache table and “aff” means that a single shared-SYMPED server is being run with process affinities set so that its processes can be partitioned similar to N-copy. A new section is added to the end of the table and it contains the results of running mpstat during the experiment. The row labelled “softirq” gives the percent of time spent servicing software interrupts. Aside from these changes, the table is similar to the profiling tables presented in the previous chapter. For each server, only the values where there is a significant difference among the servers are discussed.

Typically, more time spent in networking is an indication of higher throughput. However, the non-partitioned servers, symped-nb, sharedsymped-nb with readers/writer locks and sharedsymped-b with readers/writer locks have larger networking values (over 30%) than the other servers, which are all partitioned, without correspondingly higher throughput. Similarly, these non-partitioned servers also have large softirq values (over 60%) and large e1000 values both of which are typically associated with higher throughput. These large values show the increased overheads incurred as a result of not partitioning the processes, subnets and CPUs.

As expected, the various *µ*server versions generally have higher epoll overheads than WatPipe. As well, the blocking versions of *µ*server have larger epoll overheads than the non-blocking versions since they have significantly more processes. With WatPipe, the blocking and non-blocking versions of the server have approximately the same epoll overheads since only the polling tasks call the event mechanism and these tasks are the same between the servers. However, the N-copy WatPipe servers have lower epoll overheads than the other WatPipe servers. This result is unexpected as WatPipe has centralized event polling across all the CPUs while the N-copy versions perform separate polling in each server copy. Since event polling is throttled for the experiments being profiled, N-copy WatPipe has approximately 4 times the number of calls to epoll_wait as WatPipe. The number of calls to epoll_ctl is approximately the same. Overall, the N-copy WatPipe servers have more system calls related to events. The larger overhead for WatPipe, despite fewer overall system calls related to events, is likely a result of sharing a single epoll file descriptor across CPUs.

According to the table, the blocking *µ*server versions also incur higher scheduling overheads than the non-blocking *µ*server versions. The increase in scheduling overhead is a result of having more processes that need to be scheduled. This effect does not hold true with N-copy WatPipe. Examining the average kernel context-switching values shows that N-copy watpipe-nb has almost twice as much context switching as N-copy watpipe-b, resulting in more scheduling overhead for N-copy watpipe-nb. Since the best μserver configurations are non-blocking with only 4 processes, the amount of context switching is low so the scheduling overheads for these servers is also low. However, all the WatPipe servers perform well despite having large average context-switching per second values compared to the best performing SYMPED and shared-SYMPED servers. The WatPipe servers have 10 to 130 times more context switching but still have comparable performance, including both the non-blocking and blocking servers. While the difference in scheduling overhead among the servers can be large, the actual overhead is small. Nevertheless, the large scheduling differences highlight significant architectural differences among the servers.

The row labelled "idle" represents the amount of time the kernel spends executing its idle loop. The amount of idle time for N-copy sharedsymped-b with readers/writer locks and sharedsymped-b with readers/writer locks and process affinities is large and helps to explain their poor performance. Note as well, the large kernel+arch values for these servers. As discussed earlier, the large number of processes result in contention for both the shared cache-table and system-related data-structures. Eliminating idle time by increasing the number of processes does not work for these experiments as the memory footprint of the server becomes too large and disk I/O starts to occur. In fact, the average file-system cache-size entry for the sharedsymped-b with readers/writer locks and process affinities is smaller than the size of the file set, confirming that it incurs disk I/O during the experiment. Watpipe-nb has the lowest idle time compared to the other servers, explaining why it achieves excellent throughput despite additional overheads related to sharing data across CPUs. This advantage is realized by centralizing certain operations and not pinning all tasks to specific CPUs.

Much of the difference among the servers relates to partitioning of the server processes, subnets and CPUs. Hence, the N-copy and hybrid servers perform better than the other servers. Watpipe-nb has a peak close to the peak of the best N-copy server, N-copy symped-nb, and has a more gradual decline in performance after peak. The performance of watpipe-nb is surprising since it shares a number of queues and data structures across processes and CPUs while the N-copy servers have no sharing at the application level. As well, similar to the uniprocessor experiments, using blocking sendfile incurs a penalty compared to non-blocking sendfile for the in-memory experiments. Unfortunately, the blocking servers require a large number of kernel threads due to the high throughput of the servers, resulting in memory pressure and contention. Of the blocking servers, only the blocking WatPipe servers did not suffer from memory pressure, however, the additional contention caused by the large number of writer tasks resulted in lower performance than the non-blocking WatPipe servers. One benefit of the N-copy servers is reduced contention, resulting in better performance for N-copy watpipe-b than watpipe-b. In fact, N-copy watpipe-b performed as well as the hybrid non-blocking shared-SYMPED server, sharedsymped-nb with readers/writer locks and process affinities.

Server	**userver**	**userver**	**WatPipe**	**WatPipe**	**userver**	**userver**
Arch	**symped**	**s-symped**	**pipeline**	**pipeline**	**symped**	**s-symped**
Write Sockets	**non-block**	**block**	**non-block**	**block**	**non-block**	**non-block**
Max Conns	**100K**	**60K**	**100K**	**80K**	**80K**	**80K**
Processes/Writers	**4p**	**700p**	**12w**	**400w**	**4p**	**4p**
Other Config	**N-copy**	**N-copy,rw**	**N-copy**	**N-copy**		**rw**
Reply rate	47,474	43,378	49,229	48,116	43,329	43,308
Tput (Mbps)	5666	5180	5866	5729	5176	5171
OPROFILE DATA						
vmlinux total %	**81.70**	**84.46**	**80.78**	**81.65**	**81.38**	**80.96**
networking	29.70	20.30	27.58	28.48	33.00	32.88
memory-mgmt	30.52	24.38	28.75	29.15	27.63	27.37
file system	4.28	3.49	3.93	4.51	4.04	4.18
kernel+arch	3.91	7.48	6.45	5.85	3.97	4.02
epoll overhead	3.43	3.63	1.69	1.79	3.49	3.47
data copying	0.76	0.57	0.76	0.75	0.72	0.69
sched overhead	0.04	1.66	0.96	0.77	0.07	0.07
idle	6.6	19.72	7.39	7.18	6.15	5.96
others	2.46	3.23	3.27	3.17	2.31	2.32
e1000 total %	**11.23**	**8.71**	**10.27**	**10.45**	**11.90**	**11.92**
user-space total %	**5.33**	**4.52**	**4.7**	**3.95**	**5.01**	**5.45**
thread overhead	0.00	0	1.83	1.51	0.00	0
event overhead	1.84	1.45	0.44	0.37	1.71	1.69
application	3.49	3.07	2.43	2.07	3.3	3.76
libc total %	**0.90**	**1.19**	**1.23**	**1.10**	**0.88**	**0.84**
other total %	**0.84**	**1.12**	**3.02**	**2.85**	**0.83**	**0.83**
VMSTAT DATA						
waiting %	0	0	0	0	0	0
file-system cache (MB)	2560	2438	2360	2360	2559	2395
ctx-sw/sec (kernel)	1083	74,730	129,836	64,003	680	921
MPSTAT DATA						
softirq %	58	58	56	58	61	60

Table 4.18: Server performance statistics gathered under a load of 56,000 requests per second - 4 GB

Server Arch Write Sockets	**userver** **s-symped** **block**	**userver** **s-symped** **non-block**	**userver** **s-symped** **block**	**WatPipe** **pipeline** **non-block**	**WatPipe** **pipeline** **block**
Max Conns	**60K**	**60K**	**60K**	**100K**	**80K**
Processes/Writers	**300p**	**4p**	**600p**	**4w**	**400w**
Other Config	**rw**	**rw,aff**	**rw,aff**		
Reply rate	35,227	49,717	43,707	49,886	45,666
Tput (Mbps)	4193	5944	5210	5944	5448
OPROFILE DATA					
vmlinux total %	**82.90**	**81.94**	**83.36**	**79.33**	**80.84**
networking	30.79	28.08	21.54	29.41	28.35
memory-mgmt	22.95	31.82	24.52	28.49	27.36
file system	4.05	4.17	3.78	4.53	4.55
kernel+arch	5.79	4.27	7.15	5.62	5.79
epoll overhead	6.86	2.36	3.66	2.58	2.5
data copying	0.60	0.64	0.62	0.72	0.72
sched overhead	1.18	0.08	1.53	0.48	0.78
idle	7.3	8.2	17.28	4.53	7.66
others	3.38	2.32	3.28	2.97	3.13
e1000 total %	**10.26**	**10.82**	**9.13**	**10.91**	**10.36**
user-space total %	**4.65**	**5.49**	**4.97**	**5.44**	**4.71**
thread overhead	0	0	0.01	2.32	1.93
event overhead	1.41	1.63	1.55	0.46	0.39
application	3.24	3.86	3.41	2.66	2.39
libc total %	**1.11**	**0.89**	**1.32**	**1.16**	**1.06**
other total %	**1.08**	**0.86**	**1.22**	**3.16**	**3.03**
VMSTAT DATA					
waiting %	0	0	0	0	0
file-system cache (MB)	2415	2395	2327	2358	2360
ctx-sw/sec (kernel)	58,888	4523	72,645	11,096	59,401
MPSTAT DATA					
softirq %	64	59	57	58	56

Table 4.19: Server performance statistics gathered under a load of 56,000 requests per second - 4 GB

	Maximum Number of Connections			
Procs	40,000	50,000	60,000	70,000
4	11.71	11.23	11.17	11.03
20	12.23	12.29	×	×
32	12.45	**12.71**	×	×
40	12.42	12.52	×	×
52	12.21	×	×	×
60	12.04	×	×	×

Table 4.20: *μ*server N-copy non-blocking SYMPED experiments - 2 GB

4.7 2 GB

This section examines the effect of memory pressure on the web-server architectures for multiprocessors by configuring the server with 2 GB of memory. While the amount of memory pressure in the system is low relative to the 2.2 GB file set, the potentially high throughput of the server means the disk is a bottleneck. Only the best servers from Section 4.6 are selected for this workload. An interesting point is whether, similar to the uniprocessor experiments, the blocking servers outperform the non-blocking servers despite the fact that the system is under memory pressure and the blocking servers have a larger memory footprint.

4.7.1 Tuning N-copy *μ*server

Experiments were run to tune three versions of N-copy *μ*server: N-copy symped-nb, N-copy sharedsymped-nb with readers/writer locks and N-copy sharedsymped-b with readers/writer locks. For all versions of *μ*server, the parameters tuned are the maximum number of simultaneous connections and the number of processes.

Table 4.20 shows the results of tuning N-copy symped-nb. Each row in the table represents a different number of processes from 4 to 60, with the number of processes representing the total number of processes running across all copies of the server. The columns represent a different maximum number of connections from 40,000 to 70,000, also cumulative across all the servers. The experiments show the best performance for N-copy symped-nb is around 32 processes and 50,000 connections (12.71), with a peak throughput of 4012 Mbps occurring at 54,000 requests per second and a sustained throughput of 3856 Mbps at 70,000 requests per second. This result represents a decline of approximately 35% at peak and 32% at 70,000 requests per second compared to the best N-copy symped-nb for the 4 GB experiments.

Procs	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
4	11.60	11.28	11.13	11.07
20	12.59	13.46	×	×
40	×	13.88	×	×
60	12.70	**13.98**	×	×
80	×	×	×	×
100	12.73	×	×	×

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
200	11.95	12.45	12.00	11.24
300	12.35	**13.68**	12.76	11.95
400	12.50	13.66	12.17	×
500	12.54	12.51	×	×

(b) blocking sendfile

Table 4.21: μserver N-copy shared-SYMPED with readers/writer locks experiments - 2 GB

The vmstat data for the server for its best configuration at 70,000 requests per second shows that it has an average file-system cache-size of 1.35 GB. Despite a large file-system cache, the server spends an average of 36% of its time waiting for disk I/O. As shown in Table 4.20, increasing the number of connections results in verification problems, as well, increasing the number of processes reduces throughput and eventually causes verification errors.

As the server copies are independent, increasing the number of processes results in a large change in memory footprint. For example, increasing the number of processes from 32 to 52 with 50,000 connections at 70,000 requests per second reduces the size of the file-system cache by 64 MB and increases I/O wait to 44%. Given the high throughput of the server, it appears that disk I/O is a bottleneck despite the file-system cache containing a large portion of the file set.

No N-copy symped-b experiments are run due to its large memory footprint. Sharedsymped-nb experiments are run and its smaller memory footprint should be an advantage for this workload. Based on the results in Section 4.6, only the shared-SYMPED servers with readers/writer locks are considered for this workload.

Table 4.21(a) shows the results of tuning N-copy sharedsymped-nb with readers/writer locks. Each row in the table represents a different number of processes from 4 to 100, with the number of processes representing the total number of processes running across all copies of the server. The columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for N-copy sharedsymped-nb with readers/writer locks is around 60 processes and 50,000 connections (13.98), with a peak throughput of 4570 Mbps occurring at 54,000 requests per second and a sustained throughput of 4235 Mbps at 70,000 requests per second. This result represents a decline of approximately 26% at peak and 26% at 70,000 requests per second compared to the best N-copy symped-nb for the 4 GB experiments.

Compared to N-copy symped-nb, N-copy sharedsymped-nb with readers/writer locks has 12% higher throughput at peak and 9% higher throughput at 70,000 requests per second. The performance difference appears to be related to memory footprint. For their best configurations at 70,000 requests per second, N-copy sharedsymped-nb with readers/writer locks has approximately a 138 MB larger file-system cache on average despite having almost twice the number of processes. These two factors combine to reduce the I/O wait for N-copy sharedsymped-nb with readers/writer locks at 70,000 requests per second to 10%. However, similar to N-copy symped-nb, increasing the number of connections results in verification failures due to timeouts on large files. Since the memory footprint of the server increases as the number of processes increases, it is expected that performance should eventually start to decline. However, as each copy of the server shares a cache table, the memory footprint grows slowly. Despite this slow growth, N-copy sharedsymped-nb with readers/writer locks begins to experience verification failures beyond 60 processes. As the server processes are symmetric, the number of requests read across all server copies increases with the number of processes. Eventually, more requests are read than the server can handle and verification errors occur. As well, once the server begins experiencing timeout problems, both the number of requests read and the throughput decrease.

Table 4.21(b) shows the results of tuning N-copy sharedsymped-b with readers/writer locks. Each row in the table represents a different number of processes from 200 to 500, with the number of processes representing the total number of processes running across all copies of the server. The columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for N-copy sharedsymped-b with readers/writer locks is around 300 processes and 50,000 connections (13.68), with a peak throughput of 4589 Mbps occurring at 56,000 requests per second and a sustained throughput of 3549 Mbps at 70,000 requests per second. This result represents a decline of approximately 15% at peak and 23% at 70,000 requests per second compared to the best N-copy sharedsymped-b for the 4 GB experiments.

Despite a lower condensed area, the peak throughput of the best N-copy sharedsymped-b with readers/writer locks is approximately the same as for the non-blocking version. However, N-copy sharedsymped-b with readers/writer locks has a large decline in throughput as the request rate increases, resulting in lower overall performance. At 70,000 requests per second, N-copy sharedsymped-b with readers/writer locks has 16% lower throughput than N-copy sharedsymped-nb with readers/writer locks. Similar to the uniprocessor experiments, blocking servers require additional processes to prevent a sharp decline in throughput at higher request rates. Unfortunately, the average file-system cache for the server with 300 processes, its best configuration, is already 75 MB smaller at 70,000 requests per second than the best non-blocking configuration. Increasing the number of processes from 300 to 400 while keeping the number of connections at 50,000 further reduces the average file-system cache-size by 45 MB. At this point, any potential performance gains from the additional processes are offset by increased I/O wait due to a smaller file-system cache. While the throughput at higher request rates may increase somewhat as the number of processes increases, peak throughput declines until the overall performance of the server decreases. As for all the servers under memory pressure, there is a tension between having a sufficient number of processes to compensate for I/O wait and the increase in memory footprint resulting from the additional processes.

Unlike N-copy symped-nb and N-copy sharedsymped-nb with readers/writer locks, N-copy sharedsymped-b with readers/writer locks has fewer verification problems. Similar to the uniprocessor experiments, blocking sendfile causes the server to be self limiting because it spends more time servicing existing requests rather than reading new requests. Despite being self limiting, however, increasing the number of connections reduces the memory footprint of the server, resulting in more time spent waiting for disk I/O and eventually requests begin to timeout as the efficiency of the server decreases.

4.7.2 Tuning N-copy WatPipe

Experiments were run to tune the two versions of N-copy WatPipe: N-copy watpipe-nb and N-copy watpipe-b. For both versions of WatPipe, the parameters tuned are the maximum number of simultaneous connections and the number of writer tasks. As well, for all the WatPipe experiments for this workload, the number of reader tasks per CPU is increased from one to five. Increased concurrency with respect to reading requests is helpful when reader tasks block waiting for disk I/O related to finding and opening files.

Table 4.22(a) shows the results of tuning N-copy watpipe-nb. Each row in the table represents a different number of writer tasks from 4 to 100, with the number of writer tasks representing the total number of writers running across all copies of the server. The columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for N-copy watpipe-nb is around 100 writer tasks and 50,000 connections (14.43), with a throughput of 4594 Mbps occurring at 54,000 requests per second and a sustained throughput of 4598 Mbps at 70,000 requests per second. While peak throughput occurs at 70,000 requests per second, throughput is relatively flat after 45,000 requests per second. This result represents a decline of approximately 26% at 56,000 requests per second and 20% at 70,000 requests per second compared to the best N-copy watpipe-nb for the 4 GB experiments.

With WatPipe, the incremental cost of adding additional writer tasks is small since the address space is shared. Hence, N-copy watpipe-nb has stable performance as the number of writer tasks increases. For its best configuration at 70,000 requests per second, watpipe-nb has a file-system cache-size that is only approximately 4 MB smaller than N-copy sharedsymped-nb with readers/writer locks for its best configuration despite having 100 writer tasks compared to 60 server processes for N-copy sharedsymped-nb

Writers	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
4	11.53	10.80	10.73	10.66
20	12.46	13.82	✕	✕
40	12.52	14.40	✕	✕
60	12.51	14.32	✕	✕
80	12.51	14.33	✕	✕
100	12.52	**14.43**	✕	✕

(a) non-blocking sendfile

Writers	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
200	12.58	14.25	13.46	12.30
300	12.58	14.88	15.07	14.03
400	12.59	14.88	15.00	15.01
500	12.58	14.91	**15.25**	✕
600	12.58	14.89	✕	✕

(b) blocking sendfile

Table 4.22: N-copy WatPipe experiments - 2 GB

with readers/writer locks. While the memory footprint for both servers grows slowly as the number of writer-tasks/processes increases, the memory footprint of the N-copy watpipe-nb server grows slower. In addition, N-copy watpipe-nb does not suffer from verification problems as the number of writers increase because the tasks in WatPipe are not symmetric. Therefore, unlike adding processes to N-copy sharedsymped-nb with readers/writer locks, adding additional writer tasks to WatPipe does not result in more requests being read than can be serviced.

Increasing the number of connections has a larger effect on the memory footprint of N-copy watpipe-nb because the size of many of the data structures in WatPipe are proportional to the number of connections. Moving from 50,000 to 60,000 connections reduces the average file-system cache-size by approximately 73 MB. However, the big problem is that the server accepts too many connections resulting in verification problems due to timeouts.

Table 4.22(b) shows the results of tuning N-copy watpipe-b. Each row in the table represents a different number of writer tasks from 200 to 600, with the number of writer tasks representing the total number of writers running across all copies of the server. The columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for N-copy watpipe-b is around 500 writer tasks and 60,000 connections (15.25), with a throughput of 5012 Mbps occurring at 60,000 requests per second and a sustained throughput of 4994 Mbps at 70,000 requests per second. This result represents a decline of approximately 16% at peak and 12% at 70,000 requests per second compared to the best N-copy watpipe-b for the 4 GB experiments.

N-copy watpipe-b has 8% higher throughput than its non-blocking counterpart. This difference in performance is surprising because N-copy watpipe-b has a larger memory footprint than N-copy watpipe-nb. For example, at 70,000 requests per second, N-copy watpipe-b's file-system cache is 41 MB smaller than N-copy watpipe-nb for their respective best configurations. However, N-copy watpipe-b spends less time waiting for I/O than N-copy watpipe-nb, 3% versus 17%. Part of the reason N-copy watpipe-b spends only a small amount of time waiting for I/O is its ability to support a large number of connections without verification problems; a larger number of connections allows the server to operate more efficiently. However, even for lower connection values, where N-copy watpipe-nb also verifies, N-copy watpipe-b has higher overall throughput. For example, at 50,000 maximum connections N-copy watpipe-b has a peak of 4788 Mbps at 45,000 requests per second, 4% higher that N-copy watpipe-nb. It appears that using blocking sendfile allows for better disk efficiency and higher throughput.

N-copy watpipe-b also has higher throughput than N-copy sharedsymped-b. For their best configurations, N-copy watpipe-b has 9% higher throughput at peak than N-copy sharedsymped-b and 41% higher throughput at 70,000 requests per second. In this case, the difference is that each watpipe-b copy uses a completely shared address-space, resulting in a smaller overall memory footprint than N-copy sharedsymped-b. As seen in the non-blocking experiments, when the number of writer-tasks/processes is less than 100, the difference in memory footprint between N-copy shared-SYMPED and N-copy WatPipe is small. However, for a larger number of writer-tasks/processes the difference in memory footprint becomes important. For example, with 500 writer-tasks/processes respectively and 60,000 connections at 70,000 requests per second, N-copy sharedsymped-b has a file-system cache that is approximately 207 MB smaller on average, around 14%, compared to N-copy watpipe-b. The large performance difference at 70,000 requests per second occurs because the memory footprint of N-copy watpipe-b grows very gradually as writer tasks are added, unlike N-copy sharedsymped-b, resulting in two advantages. First, N-copy watpipe-b can support a sufficient number of writers to avoid a large decline in performance for higher request rates. Second, N-copy watpipe-b gets better performance even with fewer threads because it has a smaller memory footprint than N-copy sharedsymped-b.

4.7.3 Tuning μserver

Experiments were run to tune two versions of μserver: sharedsymped-nb with readers/writer locks and process affinities and sharedsymped-b with readers/writer locks and process affinities. For all versions of μserver, the parameters tuned are the maximum number of simultaneous connections and the number of processes. No μserver SYMPED experiments are run for this workload because μserver SYMPED does not support partitioning of requests and processes without becoming equivalent to the N-copy version.

Table 4.23(a) shows the results of tuning sharedsymped-nb with readers/writer locks and process affinities. Each row in the table represents a different number of processes from 4 to 100 and the columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for sharedsymped-nb with readers/writer locks and process affinities is around 60 processes and 50,000 connections (14.02), with a peak throughput of 4584 Mbps occurring at 56,000 re-

Procs	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
4	11.71	11.34	11.22	11.26
20	12.55	13.57	✕	✕
40	12.68	13.89	✕	✕
60	12.68	**14.02**	✕	✕
80	12.72	✕	✕	✕
100	12.70	✕	✕	✕

(a) non-blocking sendfile

Procs	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
200	11.98	12.75	12.04	11.01
300	12.46	**13.41**	12.79	11.70
400	12.58	13.36	11.76	✕
500	12.53	11.89	✕	✕
600	12.44	✕	✕	✕

(b) blocking sendfile

Table 4.23: μserver shared-SYMPED with readers/writer locks and process affinities experiments - 2 GB

quests per second and a sustained throughput of 4197 Mbps at 70,000 requests per second. This result represents a decline of approximately 23% at peak and 26% at 70,000 requests per second compared to the best sharedsymped-nb with readers/writer locks and process affinities for the 4 GB experiments.

Compared to N-copy sharedsymped-nb with readers/writer locks, sharedsymped-nb with readers/writer locks and process affinities has approximately the same throughput at peak. In fact, the performance of the two servers is similar; the differences in throughput seem to be in the range of experimental variation. At 70,000 requests per second, the average file-system cache-size of sharedsymped-nb with readers/writer locks and process affinities is approximately 5 MB smaller than N-copy sharedsymped-nb with readers/writer locks. While there is a small difference, approximately 4%, for the equivalent 4 GB experiments, disk I/O and a similar memory footprint in the 2 GB experiments seems to have equalized performance.

Table 4.23(b) shows the results of tuning sharedsymped-b with readers/writer locks and process affinities. Each row in the table represents a different number of processes from 200 to 600 and the columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for sharedsymped-b with readers/writer locks and process affinities is around 300 processes and 50,000 connections (13.41), with a peak throughput of 4576 Mbps occurring at 54,000 requests per second and a sustained throughput of 3521 Mbps at 70,000 requests per second. This result represents a decline of approximately 16% at peak and 23% at 70,000 requests per second compared to the best sharedsymped-b with readers/writer locks and process affinities for the 4 GB experiments.

Compared to N-copy sharedsymped-b with readers/writer locks, sharedsymped-b with readers/writer locks and process affinities has approximately the same throughput at peak and at 70,000 requests per second. While these two rates are similar, the N-copy server has better performance in the middle rates, resulting in better overall performance. At 70,000 requests per second, the file-system cache-size of sharedsymped-b with readers/writer locks and process affinities is approximately 81 MB smaller than N-copy sharedsymped-b with readers/writer locks. Similar to the 4 GB experiments, sharedsymped-b with readers/writer locks and process affinities has a larger memory footprint compared to the N-copy server because of larger data structures as a result of sharing file descriptors across all the server processes. Both servers have their best performance with the same parameters and both servers suffer from the same problem with this configuration, a sharp decline in throughput at higher request rates. Despite the difference in memory footprint, the performance of both servers for their best configuration at 70,000 requests per second is dominated by an insufficient number of processes, resulting in similar throughput. The effect of this larger memory footprint is more pronounced at 400 processes and beyond. For example, with 400 processes and 50,000 connections at 70,000 requests per second, the difference in memory footprint grows to 103 MB. At this point, the difference in memory footprint begins to have a large effect on performance.

4.7.4 Tuning WatPipe

Experiments were run to tune the two versions of WatPipe: watpipe-nb and watpipe-b. For both versions of WatPipe, the parameters tuned are the maximum number of simultaneous connections and the number of writer tasks.

Table 4.24(a) shows the results of tuning watpipe-nb. Each row in the table represents a different number of writer tasks from 4 to 100. The columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for watpipe-nb is around 100 writer tasks and 50,000 connections (14.43), with a throughput of 4633 Mbps occurring at 54,000 requests per second and a sustained throughput of 4565 Mbps at 70,000 requests per second. This result represents a decline of approximately 24% at peak and 22% at 70,000 requests per second compared to the best watpipe-nb for the 4 GB experiments.

Compared to N-copy watpipe-nb, watpipe-nb has approximately the same throughput at peak and at 70,000 requests per second, as well as the same condensed area. However, watpipe-nb has a smaller memory footprint. Comparing the best configuration for both servers at 70,000 requests per second, watpipe-nb has a 34 MB larger file-system cache than N-copy watpipe-nb. As well, watpipe-nb spends less time waiting for I/O, 14% versus 17% for N-copy watpipe-nb. While both of these differences are small, the expectation is that the watpipe-nb should have slightly higher throughput. However, watpipe-nb shares application data across CPUs, resulting in additional execution costs.

Watpipe-nb has better overall performance than sharedsymped-nb with readers/writer locks and process affinities. While watpipe-nb only has equivalent performance at peak, and 9% better performance at 70,000 requests per second. Since all the tasks in watpipe-nb share the same address space, it has a smaller

Writers	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
4	11.53	11.13	10.92	10.89
20	12.50	14.21	✕	✕
40	12.53	14.28	✕	✕
60	12.53	14.28	✕	✕
80	12.53	14.38	✕	✕
100	12.52	**14.43**	✕	✕

(a) non-blocking sendfile

Writers	Maximum Number of Connections			
	40,000	50,000	60,000	70,000
200	12.56	14.26	13.74	12.58
300	12.59	14.92	15.12	14.79
400	12.57	14.94	15.19	14.88
500	12.57	14.86	**15.25**	15.09
600	12.56	14.86	15.14	15.09
700	12.57	14.81	15.12	15.02

(b) blocking sendfile

Table 4.24: WatPipe experiments - 2 GB

memory footprint than sharedsymped-nb with readers/writer locks and process affinities. For their best configurations at 70,000 requests per second, watpipe-nb has an average file-system cache-size that is 36 MB larger than sharedsymped-nb with readers/writer locks and process affinities, despite having 100 writer tasks versus 60 processes for the shared-SYMPED server.

Table 4.24(b) shows the results of tuning watpipe-b. Each row in the table represents a different number of writer tasks from 200 to 700. The columns represent a different maximum number of connections from 40,000 to 70,000. The experiments show the best performance for watpipe-b is around 500 writer tasks and 60,000 connections (15.25), with a throughput of 4992 Mbps occurring at 54,000 requests per second and a sustained throughput of 4863 Mbps at 70,000 requests per second. This result represents a decline of approximately 11% at peak and 10% at 70,000 requests per second compared to the best watpipe-b for the 4 GB experiments.

Compared to N-copy watpipe-b, watpipe-b has approximately the same throughput at peak but 3% lower throughput at 70,000 requests per second. Watpipe-b has higher throughput for some of the middle rates, resulting in similar overall performance. However, watpipe-b has a smaller memory footprint than N-copy watpipe-b. Comparing the best configuration for both servers at 70,000 requests per second, watpipe-b has a 34 MB larger file-system cache than N-copy watpipe-b, the same difference as the non-blocking WatPipe servers. While this difference is small, the expectation is that watpipe-b should have slightly higher throughput. Again, any performance gains are offset due to higher execution costs resulting from sharing data across CPUs.

Watpipe-b has better overall performance than sharedsymped-b, 9% higher throughput at peak and 38% higher throughput at 70,00 requests per second. The difference in performance is due to a smaller memory footprint for watpipe-b compared to sharedsymped-b with readers/writer locks and process affinities. For their best configurations at 70,000 requests per second, watpipe-b has an average file-system cache that is 214 MB larger, approximately 16%, than sharedsymped-b with readers/writer locks and process affinities. Similar to the situation with N-copy, watpipe-b has two advantages over sharedsymped-b with readers/writer locks and process affinities. Watpipe-b has a smaller memory footprint so it can support a sufficient number of threads to prevent a decline in performance at high request rates and it can get better performance with fewer threads because its has a smaller memory footprint than sharedsymped-b with readers/writer locks and process affinities.

4.7.5 Server Comparison

Figure 4.4 presents the best performing configuration for each server-architecture implementation: *μ*server N-copy non-blocking SYMPED, *μ*server N-copy non-blocking shared-SYMPED with readers/writer locks, *μ*server N-copy blocking shared-SYMPED with readers/writer locks, N-copy non-blocking WatPipe, N-copy blocking WatPipe, *μ*server non-blocking shared-SYMPED with readers/writer locks and process affinities, *μ*server blocking shared-SYMPED with readers/writer locks and process affinities, non-blocking WatPipe and blocking WatPipe. The legend in Figure 4.4 is ordered from the best performing server at the top to the worst at the bottom. Peak server throughput varies by about 25% (4012–5012 Mbps), a range of 1000 Mbps. Without N-copy symped-nb, the difference reduces to 10% (4570–5012 Mbps), a range of 442 Mbps.

Table 4.25 ranks the performance of the servers for the 4 GB workload. Again, based on a total of three runs for each server, Tukey's Honest Significant Difference test is used to differentiate the servers with a 95% confidence level. The servers are then ranked based on mean area.

The top performing servers are N-copy watpipe-b and watpipe-b, which have the same overall performance. Though N-copy watpipe-b has better sustained throughput than watpipe-b, it is less stable after peak; the throughput of watpipe-b is more stable after peak resulting in approximately the same overall performance, despite lower sustained throughput. The next grouping of servers have approximately the same peak throughput of 4600 Mbps, but differing performance after saturation. N-copy watpipe-nb and watpipe-nb have approximately the same performance with only a small decline in throughput after peak. Similarly, N-copy sharedsymped-nb with readers/writer locks and sharedsymped-nb with readers/writer locks and process affinities have approximately the same performance but a larger decline in throughput after peak compared to non-blocking WatPipe. The final two servers in that grouping, N-copy sharedsymped-b with readers/writer locks and sharedsymped-b with readers/writer locks and process affinities, also have similar performance with a large drop in throughput after peak. Though N-copy symped-nb has the worst overall performance, its throughput after peak declines gradually and its throughput beyond 60,000 requests per second is actually higher than the two blocking shared-SYMPED servers.

Comparing the performance of the best version of WatPipe and the best version of *μ*server, N-copy

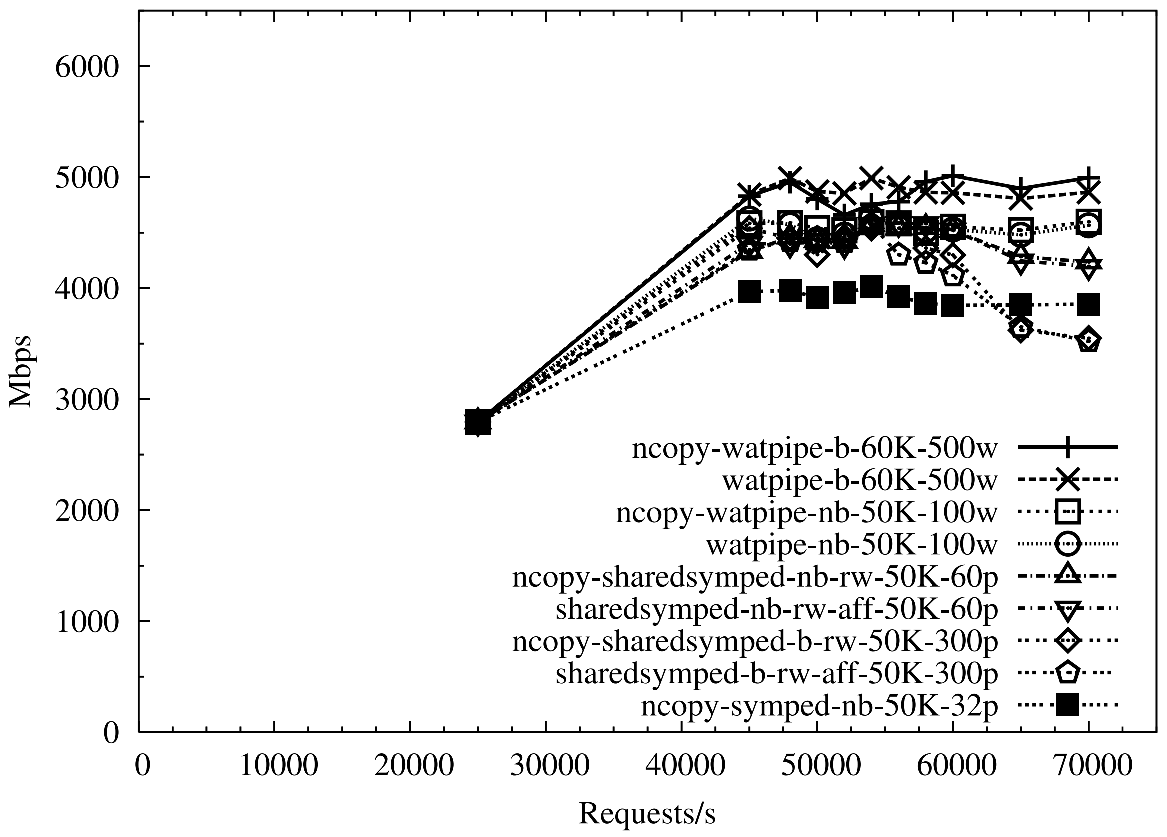


Figure 4.4: Throughput of different architectures - 2 GB

Server	Rank
N-copy watpipe-b	1
watpipe-b	1
N-copy watpipe-nb	2
watpipe-nb	2
N-copy sharedsymped-nb with readers/writer locks	3
sharedsymped-nb with readers/writer locks and process affinities	3
N-copy sharedsymped-b with readers/writer locks	4
sharedsymped-b with readers/writer locks and process affinities	4
N-copy symped-nb	5

Table 4.25: Ranking of server performance - 2 GB

watpipe-b has 9% higher throughput than sharedsymped-nb with readers/writer locks and process affinities at peak and 19% higher throughput after saturation at 70,000 requests per second. Between the best non-N-copy version of WatPipe and the best non-N-copy version of *μ*server, watpipe-b has 9% higher throughput than sharedsymped-nb with readers/writer locks and process affinities at peak and 16% higher throughput at 70,000 requests per second. For this workload, N-copy watpipe-b is the best performing server and N-copy watpipe-nb is the best performing non-blocking server. At peak, the throughput of N-copy watpipe-b is 9% higher than N-copy watpipe-nb and 9% higher at 70,000 requests per second. As mentioned earlier, the best N-copy server, N-copy watpipe-b, and the best non-N-copy server, watpipe-b, have similar throughput at peak and watpipe-b has 3% lower throughput at 70,000 requests per second. The best N-copy *μ*server, N-copy sharedsymped-nb, and the best non-N-copy version of *μ*server, sharedsymped-nb with readers/writer locks and process affinities also have approximately the same performance.

To better understand the performance of the servers, the best configuration of each server is profiled. The OProfile, vmstat and mpstat data for these experiments are summarized in Tables 4.26 and 4.27. Additional vmstat data is presented for this workload; the row labelled “blocks in” gives the average number of blocks read in per second. Otherwise, the tables are similar to the profiling tables presented in the previous workload. Note, a non-zero I/O wait value indicates that the profiling data must be scaled because the profiling data gathered only accounts for time when the CPU is executing, so it does not include I/O wait (similar to Section 3.13). For each server, only the values where there is a significant difference among the servers are discussed. The effect of running OProfile is larger on the non-N-copy servers, so their throughput is more adversely affected than the N-copy servers.

Similar to the other experiments in this thesis, for partitioned servers, higher throughput corresponds to larger networking, e1000 and softirq values, after scaling for I/O wait. The two non-blocking WatPipe servers, however, have slightly lower values than expected given their throughput. The softirq and networking values are approximately 10% lower than expected compared to the other servers. It is unclear why these values are different for the non-blocking WatPipe servers.

A larger number of threads corresponds to more context switching and higher scheduling values for a server. However, the amount of context switching is also much higher for these experiments, compared to the 4 GB workload, because the best configuration for each server tends to require more kernel threads to deal with blocking disk-I/O and kernel threads blocking waiting for disk-I/O also increase the amount of context-switching. As well, the kernel idle values for this workload are generally larger compared to the 4 GB workload. The increased idle time is likely a result of additional contention and locking in the kernel related to updating the file-system cache as file data is added or removed. These values, however, are comparable to the 4 GB experiment values for the servers that incurred disk-I/O.

The WatPipe servers have approximately 25% higher user-space execution compared to the N-copy

WatPipe servers. Increased execution time for the non-N-copy WatPipe servers is expected as application data-structures are shared across the CPUs, resulting in additional overheads, e.g., locking and cache coherency. Since the amount of time spent in user-space is small, the absolute differences are also small.

Similar to the uniprocessor experiments, the experiments for this workload show a correlation between file-system cache-size and the throughput of a server. Equivalently configured servers with smaller memory footprints tend to have higher throughput and tend to have their best configurations with more kernel threads and/or connections. However, the experiments also show an advantage to using blocking sendfile. With the uniprocessor experiments, it is difficult to separate the advantages of using blocking sendfile from the effects of the read-ahead problem in the version of the Linux kernel used for those experiments, but the patched Linux kernel used for these experiments does not have this problem. The variability of the average blocks-in per second is less for these experiments, especially if the effects of OProfile are eliminated. The mpstat data for the same experiments without OProfile running (not in the table) show an average blocks-in per second difference of 1948 (28,213–30,161) across all the servers at a request rate of 56,000 requests per second. The minor differences in blocks-in among the servers appear related to the size of the file-system cache; servers with larger file-system caches tend to have smaller blocks-in values. For example, the non-blocking and blocking WatPipe servers have average blocks-in per second values within 1% of each other, and average file-system caches size differences within 2–3%. Despite 400 additional workers, the blocking servers have average file-system caches that are only 34–43 MB smaller because the extra memory overhead is limited to each worker's stack.

At 56,000 requests per second, the blocking sendfile servers in the experiment spend less time waiting for disk I/O than the corresponding non-blocking servers. However, since the blocking servers have larger memory footprints, the expectation is they should have higher I/O wait. Part of the lower I/O wait 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 the workload tested. Comparing the disk request patterns for N-copy watpipe-nb and N-copy watpipe-b reveals some interesting observations. The blocking server makes disk-I/O requests on fewer distinct files and for the larger files these requests tend to be contiguous, allowing the server to take advantage of page cache read-ahead. The result is the blocking server makes fewer disk requests overall compared to the non-blocking server and is able to service more requests. The remainder 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. With the blocking server, a single kernel thread blocks trying to send an entire file, so the time over which the file data for a single file is accessed is small. This approach has the added advantage of reducing verifications problems due to timeouts. With the non-blocking server, a kernel thread may interleave the sending of a large file with the sending of file data for many other files, so the time over which the file data for a single file is accessed could be much longer. When there is memory pressure in the system, the kernel evicts pages from the file-system cache using an LRU algorithm. For the blocking server, since the pages associated with a file are accessed within a short span of time, it is likely that the kernel would subsequently evict the entire file from the file-system cache because these pages would become least recently used at approximately the same time. For the non-blocking server, the kernel is more likely to evict portions of various files based on the non-blocking server's interleaved access-pattern. As a result, the blocking server tends to read entire files from disk into the file-system cache when disk-I/O is necessary, while the non-blocking server tends to read in smaller portions of many files. The blocking server benefits from the efficiency of contiguous disk reads and better file read-ahead caching. A possible consequence of this difference is that the non-blocking server may read in file data from disk that is already in the file-system cache because individual disk reads tend to be for a fixed size of data. Overall, the blocking server has better disk efficiency than the non-blocking server. This difference in disk access allows a blocking sendfile server to have the same or higher throughput than the corresponding non-blocking server despite having a larger memory footprint.

The same analysis applies to the uniprocessor kernel in the previous chapter, which also has a read-ahead problem (see Section 3.13). However, both these problems benefit blocking sendfile, making it difficult to separate out the effect of each.

Server Arch Write Sockets	**userver** **symped** **non-block**	**userver** **s-symped** **non-block**	**userver** **s-symped** **block**	**WatPipe** **pipeline** **non-block**	**WatPipe** **pipeline** **block**
Max Conns Processes/Writers Other Config	**50K** **32p** **N-copy**	**50K** **60p** **N-copy,rw**	**50K** **300p** **N-copy,rw**	**50K** **100w** **N-copy**	**60K** **500w** **N-copy**
Reply rate Tput (Mbps)	31,365 3747	37,524 4490	39,097 4675	37,902 4511	41,161 4905
OPROFILE DATA					
vmlinux total %	**86.49**	**83.74**	**81.43**	**82.50**	**81.37**
networking	20.26	24.15	23.35	22.16	24.58
memory-mgmt	25.64	26.87	25.71	26.42	25.93
file system	2.96	3.34	3.86	3.28	4.05
kernel+arch	5.65	5.29	6.52	7.47	7.48
epoll overhead	1.7	2.07	3.51	1.34	1.47
data copying	0.45	0.53	0.63	0.52	0.63
sched overhead	0.14	0.39	1.69	0.99	1.48
idle	27.75	18.82	12.65	17.05	11.84
others	1.94	2.28	3.51	3.27	3.91
e1000 total %	**8.02**	**9.07**	**9.60**	**8.81**	**9.49**
user-space total %	**3.13**	**4.14**	**4.98**	**3.9**	**4.18**
thread overhead	0.00	0.00	0.00	1.74	1.71
event overhead	1.02	1.31	1.62	0.30	0.35
application	2.11	2.83	3.36	1.86	2.12
libc total %	**0.63**	**0.74**	**1.21**	**0.90**	**1.02**
other total %	**1.73**	**2.31**	**2.78**	**3.89**	**3.94**
VMSTAT DATA					
waiting %	38	22	13	23	7
file-system cache (MB)	1384	1539	1446	1519	1476
blocks-in/sec	28,593	28,442	31,614	29,620	30,399
ctx-sw/sec (kernel)	9476	34,044	179,025	128,876	208,602
MPSTAT DATA					
softirq %	37	47	48	43	49

Table 4.26: Server performance statistics gathered under a load of 56,000 requests per second - 2 GB

Server	**userver**	**userver**	**WatPipe**	**WatPipe**
Arch	**s-symped**	**s-symped**	**pipeline**	**pipeline**
Write Sockets	**non-block**	**block**	**non-block**	**block**
Max Conns	**50K**	**50K**	**50K**	**60K**
Processes/Writers	**60p**	**300p**	**100w**	**500w**
Other Config	**rw,aff**	**rw,aff**		
Reply rate	38,253	38,139	37,489	39,951
Tput (Mbps)	4560	4562	4461	4756
OPROFILE DATA				
vmlinux total %	**83.35**	**81.81**	**81.40**	**80.24**
networking	24.18	22.68	22.29	24.27
memory-mgmt	27.85	25.43	24.87	24.63
file system	3.41	3.81	3.36	4.03
kernel+arch	5.23	6.60	7.77	7.97
epoll overhead	2.09	3.39	1.78	2.09
data copying	0.54	0.59	0.56	0.62
sched overhead	0.42	1.60	1.23	1.79
idle	17.29	14.29	16.08	10.78
others	2.34	3.42	3.46	4.06
e1000 total %	**9.13**	**9.38**	**8.83**	**9.38**
user-space total %	**4.41**	**4.94**	**4.69**	**5.08**
thread overhead	0	0	2.17	2.27
event overhead	1.37	1.55	0.35	0.37
application	3.04	3.39	2.17	2.44
libc total %	**0.78**	**1.15**	**0.93**	**1.03**
other total %	**2.33**	**2.72**	**4.15**	**4.27**
VMSTAT DATA				
waiting %	21	17	20	5
file-system cache (MB)	1530	1364	1556	1522
blocks-in/sec	28,523	31,748	27,908	27,553
ctx-sw/sec (kernel)	37,632	165,682	153,640	222,268
MPSTAT DATA				
softirq %	47	45	42	49

Table 4.27: Server performance statistics gathered under a load of 56,000 requests per second - 2 GB

4.8 Comparison Across Workloads

This section examines the performance of the best multiprocessor servers in the chapter across the two workloads tested. Similar to the uniprocessor experiments, as memory pressure increases, the throughput of the servers decrease. However, with the multiprocessor experiments, even a moderate amount of memory pressure results in a sharp drop in throughput.

Figure 4.5 graphs the throughput of the servers versus the system memory-size across the two workloads at 56,000 requests per second. Unfortunately, 56,000 requests per second does not represent the peak throughput for every server, making comparisons difficult. For example, both N-copy watpipe-b and watpipe-b have approximately the same throughput at peak for the 2 GB workload, but peak at different request rates. To allow for a better comparison, Figure 4.6 graphs the condensed area of the servers across the request-rates tested for the two workloads using a stacked histogram. The servers are on the horizontal axis, with each server represented by a single bar and the corresponding condensed area is on the vertical axis. The black portion is the condensed area for the 2 GB workload and the gray portion represents the additional condensed area, due to higher throughput, for the 4 GB workload. While the condensed area gives a better picture of overall performance, it de-emphasizes differences in peak throughput.

The general pattern is the same for both the uniprocessor and multiprocessor experiments: as the memory pressure in the system increases, the throughput of the servers decrease. Therefore, the servers achieve their best performance with the 4 GB workload. When there is no memory pressure, another pattern emerges: the non-blocking servers require few kernel threads, resulting in low overheads and high throughput, and the blocking servers require more kernel threads, resulting in higher overheads and lower throughput.

The performance of all the servers drop with the 2 GB workload. When there is memory pressure in the system, a different pattern emerges: memory footprint and disk efficiency are two important factors determining server performance. The blocking version of a server benefits from better disk efficiency but it also has a larger memory footprint than the corresponding non-blocking server. Since WatPipe uses a shared address-space it can scale to a large number of kernel threads with only a small increase in memory footprint. Hence, the blocking WatPipe servers perform the best. Unfortunately, shared-SYMPED does not scale as efficiently, resulting in the blocking shared-SYMPED servers having a large memory footprint and lower performance compared to blocking WatPipe. As well, less efficient scaling means the blocking shared-SYMPED servers are unable to support a sufficient number of processes, resulting in lower overall performance than all the non-blocking servers, except N-copy symped-nb. N-copy symped-nb has the worst performance as it has even less efficient scaling and does not benefit from better disk efficiency.

Two additional factors affecting the throughput of the multiprocessor experiments are high throughput and the presence of multiple CPUs. These factors combine to affect the performance of the servers in

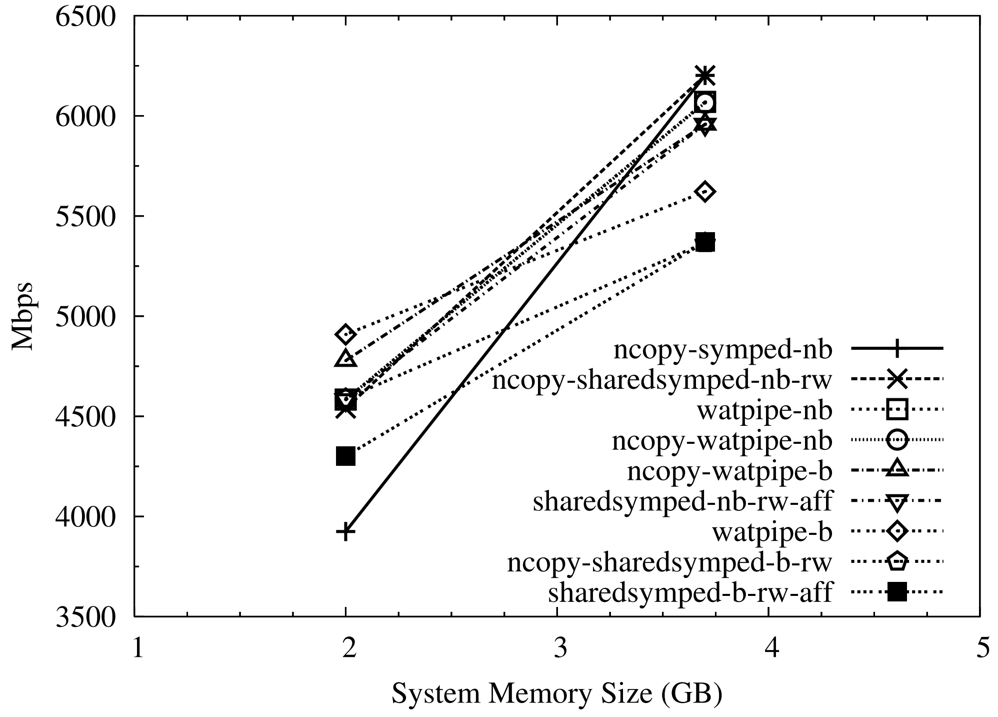


Figure 4.5: Comparison of server throughput at 56,000 requests per second across workloads

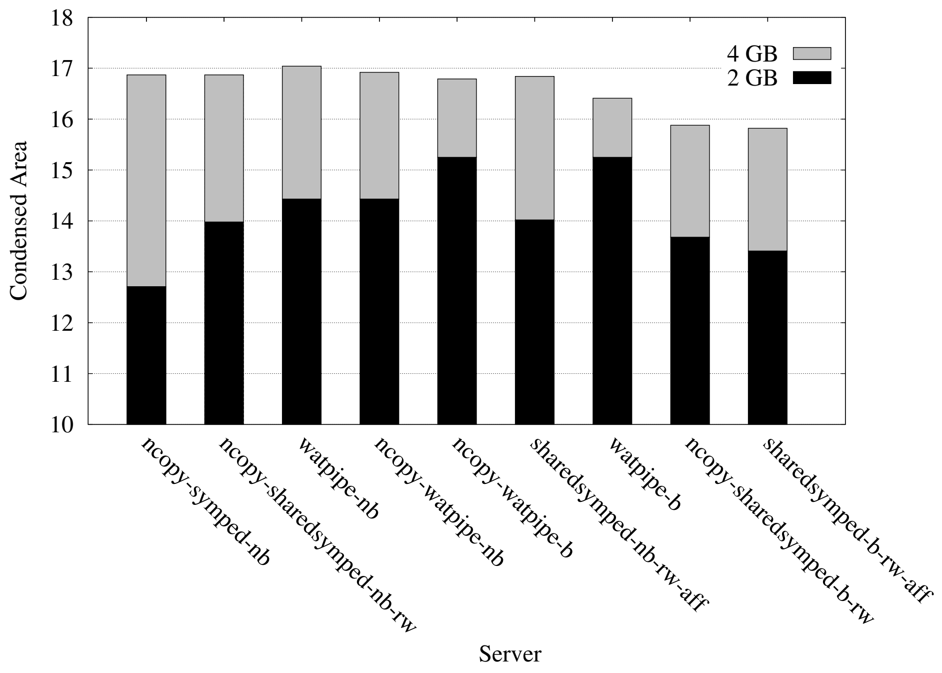


Figure 4.6: Comparison of server performance across workloads

various ways.

To achieve high throughput, the multiprocessor servers require larger configuration parameters, resulting in larger memory footprints. Specifically, the blocking servers require a large number of kernel threads to handle blocking network-I/O. While the non-blocking servers also require additional kernel threads to handle blocking disk-I/O and multiple CPUs, the increase is much smaller than for the blocking servers. Even with the 4 GB workload, N-copy sharedsymped-b and sharedsymped-b with readers/writer locks and process affinities have large enough memory footprints to affect their performance. The best configuration parameters for these two servers is at the point where disk I/O begins to occur. Hence, even though there is little to no disk I/O for the best configuration of the two servers, the memory footprint of the server limits performance; once disk I/O occurs, the performance of a server begins to level off and then decrease as its tuning parameters are increased.

To deal with blocking disk-I/O, the number of kernel threads required increases for the 2 GB workload, resulting in additional memory pressure. However, the larger memory footprints are mitigated by a larger system memory-size, 2 GB versus a system memory-size of 1.4 GB or .75 GB for the uniprocessor experiments with memory pressure. In fact, the multiprocessor servers need larger file-system caches-sizes than either of the two uniprocessor workloads with memory pressure because of higher throughput. Despite the larger file-system caches, however, the decline in throughput of the multiprocessor experiments is sharper. The presence of I/O wait indicates the 2 GB workload is disk bound. Given the high request rates, even a small percentage of requests requiring disk I/O means the disks become a bottleneck.

Another difference with the multiprocessor experiments is data sharing and contention across CPUs, reducing the parallel execution of the system, especially for the non-N-copy servers since they also share application data among a larger number of threads and across CPUs. This problem affects the in-memory experiments as well. Note the poor scalability moving from 1 CPU to 4 CPUs in Section 4.4. For example, data sharing and contention at the application level across CPUs results in worse performance for watpipe-b in the 4 GB workload compared to N-copy watpipe-b, and watpipe-b, despite having a smaller memory footprint, has performance equivalent to N-copy watpipe-b in the 2 GB workload. The problem is worse when there is disk I/O because the file-system cache is shared, resulting in additional locking and contention as file data is added and removed. As expected, better disk efficiency and a smaller memory footprint result in more efficient use of the file-system cache and lower additional overheads. Therefore, the blocking WatPipe servers receive an additional performance advantage.

The multiprocessor tuning parameters to achieve the best performance for the various servers are larger than for the uniprocessor experiments. The changes in processes/kernel-threads over the two workloads is discussed earlier in this section. To achieve high throughput, the servers must also support a large number of connections. Similar to the uniprocessor experiments, as the memory pressure in the system increases, the throughput of the servers decreases and the number of simultaneous connections required also decreases. Interestingly, for the multiprocessor experiments, the non-blocking servers achieve their best performance with a larger number of connections than the blocking servers for the in-memory workload. It is not apparent why this tuning is opposite from the uniprocessor experiments. For the 2 GB workload, the blocking and non-blocking shared-SYMPED servers have the same number of connections and the blocking WatPipe servers require a larger number of connections. It is the larger memory footprint of the blocking shared-SYMPED servers that limits their throughput, and hence, fewer connections are required.

4.9 Summary

This chapter examines the performance of several server architectures on a multiprocessor machine for two workloads: in-memory and disk bound. The architectures include SYMPED, shared-SYMPED and pipeline. Similar to the uniprocessor experiments, μserver is used for SYMPED and shared-SYMPED and a new version of WatPipe is implemented for the pipeline architecture.

The uniprocessor experiments identified a number of factors affecting the performance of the servers, including memory footprint and disk efficiency. These factors are important, however, executing on a multiprocessor introduces a number of challenges over executing on a uniprocessor. Key issues include data sharing across CPUs, affinities and partitioning, locking and high throughput. These factors must be considered when designing a web server for execution on a multiprocessor. Simply running a server from the previous chapter on a multiprocessor does not achieve high throughput, despite the presence of multiple kernel threads. For the experiments in the thesis, regardless of the server architecture or implementation, it is important to set affinities to equally distribute network interrupt processing across the available CPUs. Based on this arrangement, extending the uniprocessor servers to support partitioning is necessary to achieve best performance.

The most straightforward approach to achieve partitioning is to use an N-copy approach, with one copy of a server per CPU. The system is partitioned by pinning each server copy and its associated processes to a specific CPU and having each server copy only handle requests on subnets associated with that CPU. As there is no shared data at the application level, the expectation is the non-blocking N-copy servers should perform the best for the in-memory workloads. The uniprocessor servers are also extended to support partitioning using a hybrid approach whereby the processing of requests is partitioned but application data is shared across CPUs to varying degrees.

The advantage of sharing data is a smaller memory footprint, however, sharing data results in additional overheads to ensure safe access and can inhibit parallel execution with multiple CPUs. Despite improved locking, as the number of kernel threads sharing data increases, these overheads also increase.

The N-copy servers typically have larger memory footprints than the non-N-copy servers due to data duplication across the copies; however, the difference in memory footprint is proportional to the number of CPUs. While there may be many processes/kernel-threads, there are only a few CPUs, limiting the amount of additional duplication, e.g., the application cache is only 6 MB, so one application cache instead of four is a savings of only 18 MB. When the system is not under memory pressure, increased contention as the number of kernel threads increases results in lower throughput for the blocking non-N-copy servers. When there is memory pressure in the system, the tradeoff between memory footprint and contention results in approximately equivalent performance for the N-copy and non-N-copy versions of a server, except for blocking shared-SYMPED, where the N-copy version actually has a smaller memory footprint than the non-N-copy version due to an implementation issue.

Blocking *μ*server N-copy shared-SYMPED has a smaller memory footprint than the blocking non-N-copy version because in the presence of a large number of shared file-descriptors and a large number of processes, its index data-structures are costly enough to negate any memory savings generated for the non-N-copy servers. In fact, several of the performance differences observed are related to implementation rather than architecture. Another example is the use of a global mutex lock for the cache table. Better performance is gained by switching to two-tiered locking. A final example is the scheduling of asymmetric tasks in WatPipe has a large effect on performance.

The difference between server architectures is more pronounced than the difference between the N-copy and non-N-copy versions of a server. Again, no single server or tuning achieves the best performance across all the workloads. For the in-memory workloads, all the non-blocking servers that support partitioning have high performance and for the disk-bound workload blocking WatPipe, both N-copy and multiprocessor, has the best performance. Blocking WatPipe performs the best because of better disk efficiency due to blocking sendfile and a smaller memory footprint due to asymmetric tasks and a shared address space. In fact, N-copy blocking WatPipe performs well across both workloads, it has peak throughput within 4% of the best server for the in-memory workload and 9% higher throughput than the other servers, not including watpipe-b, for the disk-bound workload. Hence, N-copy blocking WatPipe is a good server choice across workloads.

Predicting the best server as the number of CPUs increases is difficult because other factors, such as memory size and workload, must also be considered. However, server architecture becomes more important as throughput and the number of CPUs increase. For in-memory workloads, it is likely that the non-blocking N-copy servers would have the best performance. As seen with the blocking servers, a larger number of kernel threads results in increased contention and eventually lower throughput. When there is memory pressure in the system, the situation is more difficult to predict as there is a tradeoff in the servers designs between memory footprint and contention. For situations where N-copy servers are inappropriate, the pipeline architecture appears to offer better scalability because of its use of a shared address space and asymmetric tasks. As well, the servers are also more sensitive to memory pressure as throughput increases, meaning that high disk-throughput is essential for good performance.

Chapter 5

Lessons Learned

A large part of the work for this thesis involved implementing or augmenting web servers, debugging performance problems and running experiments. While the results of this work are discussed in the previous chapters, the process of performing the work is not discussed. This chapter discusses some of my experiences and may provide useful information to anyone else undertaking similar research.

5.1 Implementing Web Servers

The main software development effort for the thesis is the WatPipe server, both the uniprocessor and multiprocessor versions. However, I also augmented μserver, Knot and Capriccio to enhance the servers and fix bugs. Some of the design and implementation choices for the various servers are discussed in this section, along with some of the implications of these choices.

It is reasonable for an application to avoid holding a lock while performing blocking I/O, unless the lock is explicitly protecting the I/O operation. However, blocking I/O can occur in subtle ways, for example, calls to open and fstat access the file system and are potentially blocking. In the uniprocessor experiments, the application cache-table in μserver shared-SYMPED is protected by a global mutex-lock. Originally, as part of adding an entry to the application cache-table, both open and fstat were called with the cache lock acquired. Since μserver shared-SYMPED has a shared application-cache, this behaviour reduces throughput when there is memory pressure because if one process blocks holding the cache lock, while waiting for I/O as a result of one of these calls, other processes would tend to block waiting on the cache lock. In fact, the problem was discovered because the server always had a small amount of I/O wait for the moderate disk-I/O experiments. To fix the problem, the cache lock is released before calling open and fstat, and reacquired afterwards. While there is extra locking overhead, the overall result is higher throughput and I/O wait is eliminated because of the additional concurrency.

A more general observation is that small decisions can become important when scaling to thousands of connections and hundreds of processes. One example is the index data-structures used in μserver to map between a socket descriptor and the array containing its corresponding request information. As discussed in Section 4.6.2, when scaling to hundreds of processes with a large number of connections, this data structure can occupy hundreds of megabytes of space.

All the servers tend to operate more efficiently when a larger number of events are returned from each call to the event mechanism. An interesting observation is that more events returned for one call to the event mechanism leads to more events per call for subsequent calls to the event mechanism. When more events are returned from a call, more work is performed processing the events, so the processes/kernel-threads tend to execute longer resulting in a larger number of events available for the next call.

While these observations apply to all the servers, there are a number of lessons learned specifically related to the design and implementation of WatPipe.

5.1.1 WatPipe

One of the major design features of WatPipe is the use of asymmetric tasks. Asymmetric tasks offer many benefits but allowing tasks to execute freely and be scheduled by the operating-system scheduler can lead to problems such as contention and poor cache behaviour. The throughput of WatPipe can vary greatly depending on how these tasks are scheduled. To deal with these problems, WatPipe uses convoy or cohort scheduling so reader tasks are not active at the same time as writer tasks. As well, the execution of the event polling tasks are also controlled by WatPipe. Because these tasks perform very little work aside from calling epoll_wait, they do not use their entire time slice, so the Linux scheduler tends to schedule them frequently. However, event polling is more efficient when the delay between polls is longer, i.e., fewer calls result in more events returned per call. With the uniprocessor server, information is centralized so deciding when to perform polling is simple. With the multiprocessor server, sections of the server are executing independently on different CPUs, so making a global decision is difficult. Hence, the simple approach of throttling the polling tasks using a small delay is used.

When designing the multiprocessor version of WatPipe, determining how to partition the server was difficult. The approach taken is to partition most of the server, including the read stage and the write stage, and creating a separate copy per CPU of the the cache table and most of the internal data-structures. In-memory experiments were then run to confirm the server's throughput is equivalent to the N-copy server and to generate a baseline throughput for subsequent comparisons. Then, parts of the system were unpartitioned in turn and experiments were run to determine the effect on performance. This systematic unpartitioning identified places where partitioning is necessary for high performance, resulting in the multiprocessor WatPipe server presented in the thesis. For example, unpartitioning the read and write stages resulted in a large drop in throughput, but the acceptor tasks and most of the internal data structures did not require partitioning. An unpartitioned application-cache also resulted in lower throughput initially, but moving from a single global lock to two-tiered locking made a shared application-cache reasonable.

The initial design of multiprocessor WatPipe used a single listening-port with a small number of acceptor tasks. However, the final design contains separate listening-ports with one acceptor task per subnet. While using a single port for accepting connections is more efficient, it does not provide enough control. Because WatPipe uses fixed-sized queues and separate per CPU queues, it is inappropriate, at times, to accept connections from any subnet; but it may be reasonable to continue accepting connections for certain subnets. Therefore, each subnet must be handled separately to control the subnets from which new connections are accepted. While it is possible to reduce the number of acceptor tasks by using accept with a non-blocking listening socket, this approach leads to polling. It is more efficient to use accept with blocking sockets, requiring a separate acceptor task per subnet.

5.2 Performance Problems

Debugging for the thesis can be classified into one of two categories: correctness and performance. Tracking down either type of problem is challenging, especially if they only occur when the server is under a full load. The focus of this section is the kernel problems discovered in the course of the experiments. While at least two of the kernel problems can be considered correctness issues, all the problems presented themselves initially as performance problems.

Debugging performance problems are difficult, especially tracking a performance problem into the Linux kernel. The first step is to recognize that a performance problem actually exists. In isolation, it is difficult to determine if a server is running reasonably or if there is a problem. One advantage of comparing multiple servers across various configurations is the opportunity to compare throughput among the servers to identify performance anomalies. Tracking down the source of an anomaly can be challenging if these anomalies only tend to occur when the server is under a full load. The first step is to determine whether the server itself is defective or if an external factor is causing the problem. If an anomaly occurs consistently with certain types of servers or server configurations, it might suggest a deeper problem. This situation occurred several times over the course of the thesis. In one case, there was a consistent performance difference between the non-blocking and blocking servers for the disk bound workloads. In another case, using a separate cache per process unexpectedly had better performance than using a shared cache. In a final example, there was inconsistent throughput for multiple runs with the same server configuration. In each of these case, the problem was eventually traced into the Linux kernel

In tracking down these problems, I found OProfile data was not helpful because it tended to be too coarse grained. Rather, other types of data gathered during the experiments, such as differences in the average blocks-in from disk or the amount of I/O wait, were more helpful. Unexpected differences in these statistics, helped to confirm a problem exists and even suggested what type of problem is occurring.

A tool I found useful in tracking down Linux kernel problems is SystemTap [2]. SystemTap is a scripting language useful for instrumenting a running Linux kernel. It works by executing a handler on specified events, such as on entry to or exit from specified kernel functions. One useful feature of SystemTap is the ability to access and print local context inside the kernel.

For example, this technique was used to track down the read-ahead problem with non-blocking sendfile in the 2.6.16-18 Linux kernel. Initially, SystemTap was used to understand and track the behaviour of sendfile into the kernel. After identifying the important functions and data structures involved, a subset of the function parameters was printed on entry to these functions. Looking specifically at the output for the functions involved in managing page-cache read-ahead revealed that read-ahead was being disabled with non-blocking sendfile. Based on this information, an examination of the relevant source code revealed the source of the anomaly, a mismatch between the amount of disk I/O and network I/O on calls to sendfile. Without a tool like SystemTap to trace the sendfile call and narrow the search space, finding the problem would have taken significantly longer because the Linux kernel is large and complicated.

5.3 Performance Experiments

There are a number of advantages to using the Linux kernel and other open-source software for performance experiments. Access to the source is invaluable in tracking down problems and understanding performance issues. A frequent suggestion when a problem is found is to upgrade to a newer version. However, the Linux kernel and open-source software, in general, are moving targets. In many cases, upgrading to a new kernel does not solve a given problem, and newer versions of the kernel can have their own problems, leading to performance regressions. For example, Figure 5.1 shows the results of multiple N-copy sharedsymped-nb with readers/writer locks experiments on the Linux kernel 2.6.24-3. The lines labelled “No patch” are two separate runs of the same experiment, without rebooting the machine in between runs, with an unpatched version of the kernel. The lines labelled “Patch” are two separate runs of the same experiment, without rebooting the machine in between runs, with a patched version of the kernel (the same kernel used for the experiments in Chapter 4, containing the patch in Section A.2). The results show that the throughput for the unpatched kernel is much lower than the patched kernel and the

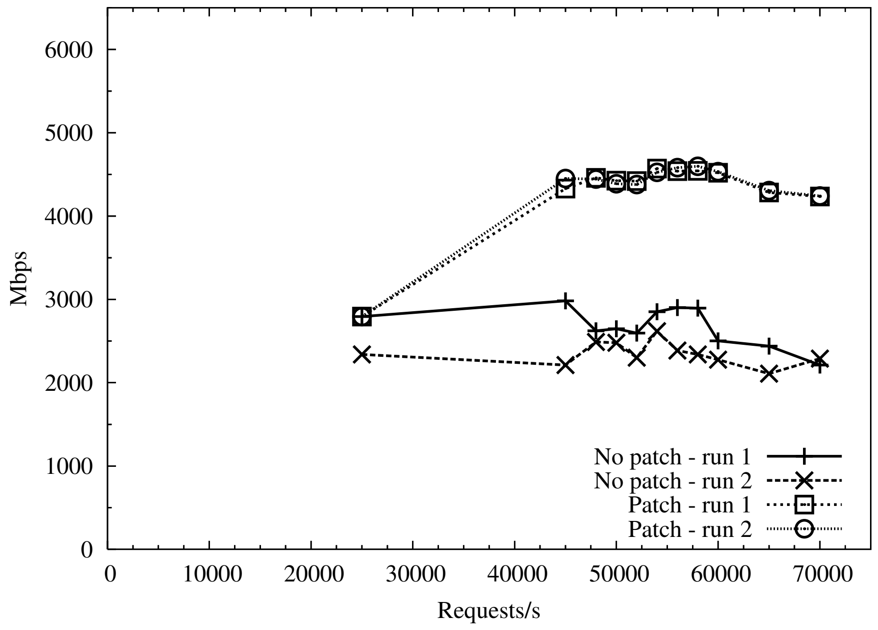


Figure 5.1: Experiments unpatched and patched Linux kernel 2.6.24-3 - 2 GB

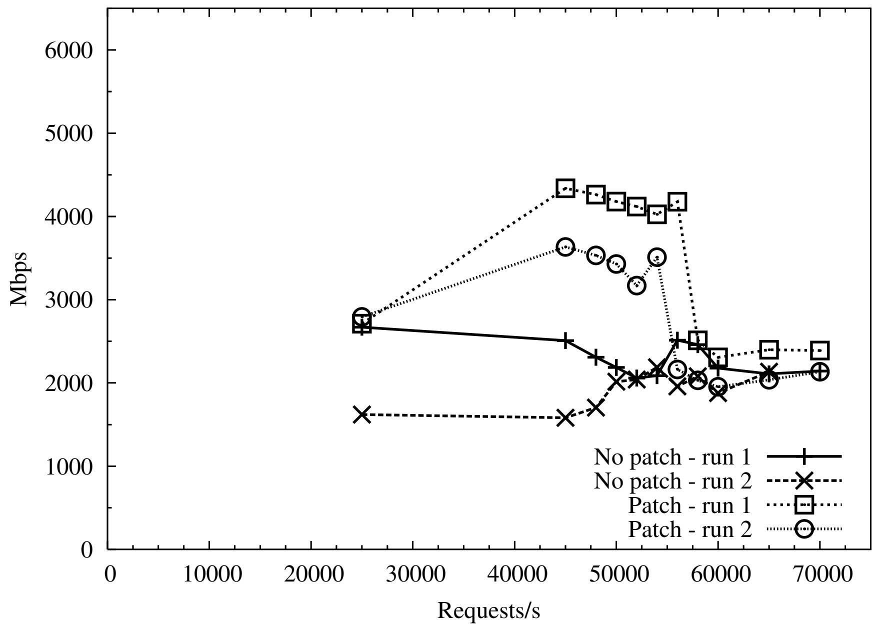


Figure 5.2: Experiments with unpatched and patched Linux kernel 2.6.32 - 2 GB

performance of the second run is worse than the first run. The runs with the patched kernel have higher throughput and the performance is approximately equivalent between runs.

Figure 5.2 contains the same sequence of experiments with a 2.6.32 Linux kernel. The unpatched kernel behaves similarly to the unpatched 2.6.24-3 kernel. With the same patch, experiments with the newer kernel also show much higher performance. However, experiments with the patched kernel have two problems. First, there is a significant drop in performance at around 54,000 requests per second. Second, the second patched run has lower throughput than the first patched run. Without further investigation, it is unclear whether the problem is with the patch or if there is another problem with the kernel. In either case, this small experiment illustrates the challenges of working with rapidly changing open-source software and the fallacy that upgrading to a new version is a fix.

Many performance experiments were run for the thesis. In fact, all of the experiments were run more than once, and in some cases many times due to various problems. The most important lesson when running a large number of experiments is to create scripts to automate the process of running experiments and summarizing the results as much as possible. Building scripts can be time consuming, but every script I wrote has been used many times and has ended up saving a large amount of time.

Chapter 6

Conclusion

This thesis examines uniprocessor and multiprocessor web-server architectures for serving static content to determine the key factors affecting their performance. The uniprocessor architectures examined are thread-per-connection, SYMPED, shared-SYMPED and pipeline. Knot and Capriccio are used for the thread-per-connection architecture, *μ*server for SYMPED and shared-SYMPED, and WatPipe for pipeline. WatPipe was implemented for the thesis and the other servers have been augmented so the implementation for each architecture is state-of-the-art and uses current best practices. The multiprocessor architectures examined are SYMPED, shared-SYMPED and pipeline. To achieve best performance on a multiprocessor, server architectures and implementations must be adjusted to support partitioning of kernel-threads, subnets and CPUs. The corresponding uniprocessor servers are extended for multiprocessor execution and also compared against N-copy versions of the servers.

Extensive experiments were run for each server on two workloads, in-memory and heavy disk-I/O, and for the uniprocessor servers a moderate disk-I/O workload was also tested. These experiments are used to compare the performance of the servers and to determine the factors that are important to achieve high throughput.

Regardless of the server architecture or implementation, proper tuning is critical to achieve best server performance. Furthermore, no single tuning achieves the best performance for all workloads. An important difference among the servers is the range of parameters over which performance is stable. The blocking servers tend to be more stable as the number of connections are increased and are less susceptible to large-file timeouts than the non-blocking servers. As well, the servers with shared memory and asymmetric tasks show better tuning stability as the number of kernel threads increase than the servers with symmetric processes. Stability is an important feature as it makes server tuning easier and it allows a server configuration to operate well over a wide range of workloads.

One interesting result is the performance difference between non-blocking and blocking sendfile across the workloads. For in-memory workloads, the most important factor is to keep the execution overhead of the server small. Hence, the non-blocking servers perform best as they require few kernel threads. Once there is memory pressure in the system, performance is dictated by the memory footprint of the server. Since a large number of kernel threads are required to handle blocking disk-I/O, servers with shared address-space have higher throughput. As the number of kernel threads increases, the use of asymmetric tasks incur less additional execution overheads compared to symmetric processes, though the differences are small. Once the server is disk bound, efficient disk access is also an important factor in determining throughput. Despite having a larger memory footprint and additional overheads, servers using sendfile with blocking sockets have better disk efficiency due to different file-access patterns, usually resulting in higher throughput with disk-bound workloads. For the multiprocessor experiments, the high throughput of the servers make memory footprint even more important. Even a moderate amount of memory pressure causes the servers to become disk bound. Unlike the uniprocessor experiments, better disk efficiency is unable to overcome large memory-footprint problems for some of the servers.

A final factor that is important for the multiprocessor servers is sharing data across CPUs. The advantage of sharing data is a smaller memory footprint, but the trade off is increased overheads due to contention and cache coherency. Any server architecture with shared application data must control these overheads, otherwise they can become a bottleneck. The use of two-tiered locking for the application cache seems to work reasonably well when there are only a few kernel threads but less well as the number of threads increase. However, the shared memory designs may more easily support other kinds of work within a server, such as dynamic monitoring, load balancing and tuning.

Comparing the performance of the servers across the workloads tested, blocking WatPipe offers the best performance among the uniprocessor servers and N-copy blocking WatPipe among the multiprocessor servers. More importantly, consider the factors that result in these servers having the best overall performance. Both servers use blocking sendfile, resulting in better disk efficiency when there is heavy disk-I/O. While all the servers have blocking versions, both servers also have a small memory footprint even with a large number of kernel threads due to shared address space. Even though N-copy blocking WatPipe does not have a completely shared address space, the amount of duplication is small as it is proportional to the number of CPUs. The trade off for the larger memory footprint is less contention as there is no sharing of application data across CPUs. Finally, both servers use asymmetric tasks, allowing the server to add tasks where needed without unnecessary overheads, resulting in high throughput even with the in-memory workload despite a large number of kernel threads.

While no single server or configuration performed the best for all workloads, the difference in peak throughput among the best version of each server architecture is within 9–13%, across the uniprocessor and multiprocessor workloads, where at least one server of each kind of architecture appears with these ranges. Unless highest throughput is critical, secondary factors may determine the appropriate choice of server architecture, e.g., tuning stability, ease of implementation and debugging, and programming preference.

6.1 Future Work

There are a number of areas for further work related to the thesis. In addition to examining alternative architectures, especially for multiprocessors, there are a number of other avenues to explore.

One of the big problems with the servers in the thesis is the need for hand tuning. Tuning is critical to server performance, however, no single tuning performs best across all workloads. Ideally, the server should dynamically adapt to changing workloads as necessary by performing auto tuning. Auto tuning, however, especially across a number of different parameters, is a difficult problem [11]. As shown in the thesis, simple heuristics like eliminating idle time are insufficient. Servers must be able to monitor their behaviour and adjust accordingly. As well, the various architectures examined in the thesis show varying degrees of sensitivity to the tuning parameters tested, especially for large tuning values. Examining server architectures specifically with respect to ease of tuning is also reasonable.

For the experiments in the thesis with heavy disk-I/O, all the servers experienced I/O wait because blocking disk-I/O becomes a bottleneck once the file-system cache is sufficiently small. Since the presence of I/O wait indicates there is extra CPU time in the system, performing extra work in order to reduce the memory footprint of the servers could result in a higher throughput. All the servers store HTTP headers in their application cache, resulting in a larger memory footprint. It would be interesting to see the effect of dynamically generating HTTP headers for each request, especially for the SYMPED servers since each process has a separate application cache, so the reduction in memory footprint is non-trivial. Other opportunities to trade additional CPU execution for memory savings may also exist.

An important result in the thesis is the performance of blocking versus non-blocking sendfile as memory pressure changes. Rather than basing this choice on workload, it may be be reasonable to dynamically adjust the behaviour of sendfile based on file size. For example, using blocking sendfile for large files and non-blocking sendfile otherwise. Experiments are required to understand the effect of this dynamic adjustment on disk access-patterns and number of kernel threads, as well as to determine an appropriate file size for this transition.

There are also interesting areas to explore specifically related to multiprocessor architectures. A first step is moving to a 64-bit operating system and testing the scalability of various server architectures with more memory and CPUs. The trend of increasing CPUs and network interface capacity make it likely that CPUs will outnumber network interfaces on newer hardware, making partitioning more challenging. Furthermore, new multiprocessor hardware may require different approaches for best performance. For example, instead of partitioning, parallelizing operating-system interrupt-handling code and network stacks may be required.

Based on working with several server architectures, a couple of small operating system improvements may help when implementing servers. One problem with some servers (WatPipe, Knot) is the inappropriate scheduling of asymmetric threads by the operating system scheduler, as its general policy is fairness. However, due to factors such as contention and cache coherency, it may be reasonable to use techniques like cohort [33] or convoy [60] scheduling, where only a subset of the ready threads are run concurrently. Unfortunately, these self-scheduling techniques can lead to inefficient CPU utilization due to potentially blocking operations, such as disk I/O, because it is unknown in advance whether an operation will block, so the application is unable to determine if it should adjust its current thread schedule. Techniques involving thread priorities tend to be awkward, difficult to control and may become expensive if priorities need constant adjusting. Therefore, a reasonable enhancement is for the operating system kernel to wake up a voluntarily blocked thread when another thread involuntarily blocks. For example, signalling a thread blocked on a condition variable or providing a special type of yield system-call to allow threads to voluntarily delay their execution until a blocking operation occurs.

Another useful extension is to allow minimums to be specified for operations like event polling. In many cases, it is preferable to wait until a number of events are available to reduce the number of event polling calls required. For example, if an application has more than one thread, polling for events may not be an indication that the application has run out of work. Specifying a minimum number of events with a timeout would still allow for the timely delivery of events while reducing the amount of polling, but additional tuning would be required to determine this minimum.

As well, once operating systems provide better asynchronous I/O support, it would be interesting to determine if these mechanisms offer any performance benefits for high-performance web-servers. Integrating asynchronous I/O with existing event mechanisms would present an application with a single consistent interface for all I/O, simplifying programming [42]. As well, asynchronous I/O could reduce the memory footprint required for certain server architectures.

Finally, it would be interesting to examine the performance of the various server architectures with different types of Internet applications and other workloads because architectures that work best for static web-servers may not work well for other application domains. For example, other types of servers, such as video streaming or game servers, and other workloads, such as dynamic workloads and web 2.0 workloads, have different requirements.

APPENDICES

Appendix A

Kernel Patches

A.1 Patch for Linux kernel 2.6.16-18

```
*** try/linux-2.6.16.18/mm/filemap.c      2006-05-22 14:04:35.000000000 -0400
--- linux-2.6.16.18-rafix/mm/filemap.c   2009-07-03 23:14:58.000000000 -0400
*************** page_ok:
*** 803,809 ****
                 * When (part of) the same page is read multiple times
                 * in succession, only mark it as accessed the first time.
                 */
!               if (prev_index != index)
                        mark_page_accessed(page);
                prev_index = index;

--- 803,809 ----
                 * When (part of) the same page is read multiple times
                 * in succession, only mark it as accessed the first time.
                 */
!               if (prev_index != index || !offset)
                        mark_page_accessed(page);
                prev_index = index;
```

A.2 Patch for Linux kernel 2.6.24-3

```
*** linux-source-2.6.24/fs/splice.c       2008-11-30 17:09:49.000000000 -0500
--- linux-source-2.6.24-cachefix/fs/splice.c      2009-07-24 11:30:07.000000000 -0400
*************** fill_it:
*** 412,417 ****
--- 412,418 ----
                if (unlikely(!isize || index > end_index))
                        break;

+               mark_page_accessed(page);
                /*
                 * if this is the last page, see if we need to shrink
                 * the length and stop
```

References

[1] The μserver home page. HP Labs, 2005. http://www.hpl.hp.com/research/linux/userver/. 7

[2] The SystemTap home page, 2010. http://sourceware.org/systemtap/. 174

[3] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association. 7, 34

[4] Vaijayanthimala Anand and Bill Hartner. TCPIP network stack performance in Linux kernel 2.4 and 2.5. In *Proceedings of the 4th Annual Ottawa Linux Symposium*, June 2002. 120

[5] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992. 21

[6] Apache software foundation. The Apache web server. http://httpd.apache.org. 10

[7] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 6–6, Berkeley, CA, USA, 1997. USENIX Association. 24

[8] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1998. USENIX Association. 19

[9] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999. 5, 19, 33

[10] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS 1998*, Madison, Wisconsin, 1998. 25

[11] Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. Understanding tuning complexity in multithreaded and hybrid web servers. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008. 28, 179

[12] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verità, Switzerland, May 2009. 10

[13] Tim Brecht, David Pariag, and Louay Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004. 8, 14, 15, 23, 35

[14] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: A language for programming high-performance servers. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 129–142, 2006. 11, 14, 17

[15] Anupam Chanda, Khaled Elmeleegy, Romer Gil, Sumit Mittal, Alan L. Cox, and Willy Zwaenepoel. An efficient threading model to boost server performance. Technical Report TR04-440, Rice University, 2004. 21

[16] Abhishek Chandra and David Mosberger. Scalability of Linux Event-Dispatch mechanisms. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001. 8, 19, 20

[17] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, and Chita R. Das. A multi-threaded pipelined web server architecture for SMP/SoC machines. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 730–739, New York, NY, USA, 2005. ACM. 2, 11, 16

[18] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963. 34

[19] Ryan Cunningham and Eddie Kohler. Making events less slippery with eel. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association. 7

[20] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 186–189, New York, NY, USA, 2002. ACM. 7

[21] Ulrich Drepper. The need for asynchronous, zero-copy network I/O. In *Proceedings of the 8th Annual Ottawa Linux Symposium*, July 2006. 21

[22] Ulrich Drepper and Ingo Molnar. The native POSIX threads library for Linux. http://people.redhat.com/drepper/nptl-design.pdf. 11

[23] Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *USENIX Annual Technical Conference, General Track*, pages 241–254, 2004. 21

[24] Annie Foong, Jason Fung, and Don Newell. An in-depth analysis of the impact of processor affinity on network performance. In *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*, volume 1, pages 244–250 vol.1, Nov. 2004. 120

[25] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast user-level locking in Linux. In *Ottawa Linux Symposium*, June 2002. 38, 128

[26] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the 6th Annual Ottawa Linux Symposium*, July 2004. 19

[27] Philipp Haller and Martin Odersky. Actors that Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, pages 171–190. Springer, 2007. 7

[28] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Global Telecommunications Conference, 1997. GLOBECOM '97., IEEE*, volume 3, pages 1924–1931 vol.3, Nov 1997. 20

[29] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John 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. 1, 14, 20

[30] Jupiter Research. Retail website performance: Consumer reaction to a poor online shopping experience. `http://www.akamai.com/4seconds`, 2006. 26

[31] Maxwell Krohn. Building secure high-performance web services with OKWS. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association. 1

[32] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association. 7

[33] James R. Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 103–114, Berkeley, CA, USA, 2002. USENIX Association. 10, 11, 12, 180

[34] Hugh C. Lauer and Roger M. Needham. On the duality of operating systems structures. In *Proceedings of the 2nd International Symposium on Operating Systems, IRIA*, October 1978. 7

[35] Jonathan Lemon. Kqueue - a generic and scalable event notification facility. In *USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001. 19

[36] Chuck Lever, Marius Eriksen, and Stephen Molloy. An analysis of the TUX web server. Technical report, University of Michigan, CITI Technical Report 00-8, Nov. 2000. 14

[37] Ren Liyong and Wang Tao. Study and implementation of a heavy server architecture. In *Wireless Communications, Networking and Mobile Computing, 2006. WiCOM 2006.International Conference on*, pages 1–4, Sept. 2006. 9, 38

[38] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 110–121, New York, NY, USA, 1991. ACM. 21

[39] David Mosberger and Tai Jin. httperf tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998. 24

[40] Erich Nahum, Tsipora Barzilai, and Dilip D. Kandlur. Performance issues in www servers. *IEEE/ACM Trans. Netw.*, 10(1):2–11, 2002. 20

[41] Jakob Nielsen. *Designing Web Usability*. New Riders, 2000. 26

[42] Michal Ostrowski. A mechanism for scalable event notification and delivery in Linux. Master's thesis, Department of Computer Science, University of Waterloo, November 2000. 180

[43] John K. Ousterhout. Why threads are a bad idea (for most purposes), January 1996. Presentation given at the 1996 USENIX Annual Technical Conference. 7

[44] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999. 2, 7, 8, 9, 10, 13, 15, 33

[45] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla. Comparing the performance of web server architectures. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 231–243, New York, NY, USA, March 2007. ACM. 15, 30, 37, 62

[46] KyoungSoo Park and Vivek S. Pai. Connection conditioning: Architecture-independent support for simple, robust servers. In *Proceedings of the Third Symposium on Networked Systems Design and Implementation (NSDI 2006)*, San Jose, CA, May 2006. 14, 23

[47] Niels Provos and Chuck Lever. Scalable network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000. 19

[48] Red Hat, Inc. *TUX 2.2 Reference Manual*, 2002. 14

[49] Amol Shukla, Lily Li, Anand Subramanian, Paul A. S. Ward, and Tim Brecht. Evaluating the performance of user-space and kernel-space web servers. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 189–201. IBM Press, 2004. 14

[50] Standard Performance Evaluation Corporation. *SPECWeb99 Benchmark*, 1999. http://-www.specbench.org/osg/web99. 23

[51] Sun Microsystems. Sun Fire X4150, X4250, and X4450 server architecture. Whitepaper, 2008, OPTnote = , OPTannote = . 140

[52] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, New York, NY, USA, 2007. ACM. 10

[53] Gautam Upadhyaya, Vijay S. Pai, and Samuel P. Midkiff. Expressing and exploiting concurrency in networked applications with Aspen. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–23, New York, NY, USA, 2007. ACM. 11, 17, 23

[54] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM. 123

[55] John Vert. Writing scalable applications for Windows NT, 1995. 19

[56] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea for high-concurrency servers. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003. 7, 35

[57] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for Internet services. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281, New York, NY, USA, 2003. ACM Press. 2, 10, 11, 14, 15, 23, 33, 34

[58] Ivan Voras and Mario Žagar. Characteristics of multithreading models for high-performance IO driven network applications. *The Computing Research Repository (CoRR)*, abs/0909.4934, September 2009. 17

[59] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press. 2, 11, 12, 13, 23, 33

[60] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A design framework for highly concurrent systems. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2000. 12, 180

[61] Nian-Min YAO, Ming-Yang ZHENG, and Jiu-Bin JU. Pipeline: a new architecture of high performance servers. *SIGOPS Oper. Syst. Rev.*, 36(4):55–64, 2002. 2, 11

[62] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 239–252, June 2003. 1, 2, 7, 8, 15, 23, 38, 131

[63] Zeus technology. Zeus web server. http://www.zeus.com/products/zws. 7

[64] George K. Zipf. *Human Behavior and the Principle of Least Effort: an Introduction to Human Ecology*. Addison-Wesley, Cambridge, MA., 1949. 24