High Performance Web Servers: A Study In Concurrent Programming Models

by

Srihari Radhakrishnan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

© Srihari Radhakrishnan 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With the advent of commodity large-scale multi-core computers, the performance of software running on these computers has become a challenge to researchers and enterprise developers. While academic research and industrial products have moved in the direction of writing scalable and highly available services using distributed computing, single machine performance remains an active domain, one which is far from saturated.

This thesis selects an archetypal software example and workload in this domain, and describes software characteristics affecting performance. The example is highly-parallel web-servers processing a static workload. Particularly, this work examines concurrent programming models in the context of high-performance web-servers across different architectures — threaded (Apache, Go and μKnot), event-driven (Nginx, μServer) and staged (WatPipe) — compared with two static workloads in two different domains. The two workloads are a Zipf distribution of file sizes representing a user session pulling an assortment of many small and a few large files, and a 50KB file representing chunked streaming of a large audio or video file. Significant effort is made to fairly compare eight web-servers by carefully tuning each via their adjustment parameters. Tuning plays a significant role in workload-specific performance. The two domains are no disk I/O (in-memory file set) and medium disk I/O. The domains are created by lowering the amount of RAM available to the web-server from 4GB to 2GB, forcing files to be evicted from the file-system cache. Both domains are also restricted to 4 CPUs.

The primary goal of this thesis is to examine fundamental performance differences between threaded and event-driven concurrency models, with particular emphasis on user-level threading models. However, a secondary goal of the work is to examine high-performance software under restricted hardware environments. Over-provisioned hardware environments can mask architectural and implementation shortcomings in software – the hypothesis in this work is that restricting resources stresses the application, bringing out important performance characteristics and properties. Experimental results for the given workload show that memory pressure is one of the most significant factors for the degradation of web-server performance, because it forces both the onset and amount of disk I/O. With an ever increasing need to support more content at faster rates, a web-server relies heavily on in-memory caching of files and related content. In fact, personal and small business web-servers are even run on minimal hardware, like the Raspberry Pi, with only 1GB of RAM and a small SD card for the file system. Therefore, understanding behaviour and performance in restricted contexts should be a normal aspect of testing a web server (and other software systems).

Acknowledgements

I thank my advisor Dr.Peter Buhr for his endless enthusiasm and knowledge. My graduate studies took a few unexpected and unusual turns, and I thank him for his support and patience through it all. I have enjoyed working with him immensely, and I will miss our conversations and collaboration. I would like to thank Ashif Harji, whose Doctoral Thesis formed the basis of my work, and Google for generously supporting my research efforts.

Dedication

To Neha, my parents, and my sister.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the past decade, there has been an exponential increase (see Figure 1.1) in the amount of data traffic web-services need to handle on a daily basis [49]. With an increasing amount of revenue for many businesses coming from the Internet, either directly through advertisements or indirectly through popularity in social media such as Facebook and Twitter, there is a greater need for faster, scalable, and reliable web services. Most modern web-services are multi-tiered (see Figure 1.2) and use commodity hardware for scalability. While application-scaling systems exist to monitor an application (web server) and automatically adjust capacity to maintain predictable performance at low cost [6], the onus is still on the developer to program for performance, scalability and availability in the presence of peak application-loads without failures. That is, an application-scaling tool cannot magically turn a poor application into a good application (garbage in, garbage out). Performance is important because a small improvement in request latency can be the difference between keeping and losing user engagement, which in turn translates into revenue impact for companies.

1.1 Software Paradigm Shift

A significant portion of business revenue goes towards scaling some form of internet access in the presence of failures [54], and is the bread-and-butter of some of the largest technology

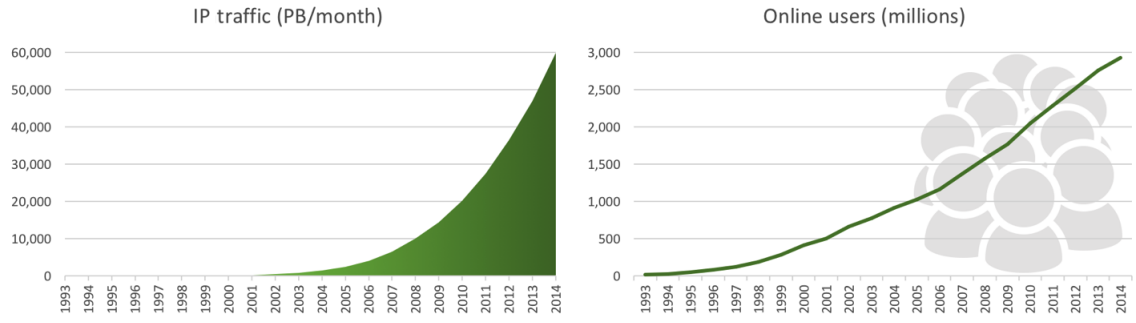


Figure 1.1: Growth of web-server traffic over the last decade

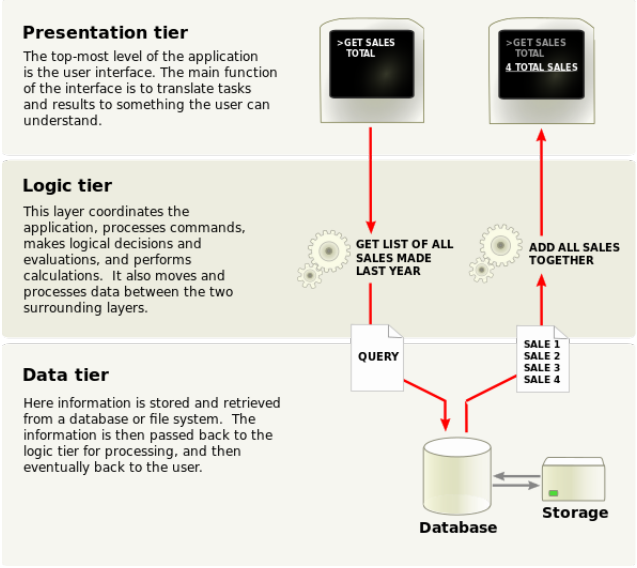


Figure 1.2: Multi-Tiered Web-Architecture

companies in the world. These systems have evolved a thousand fold in a decade, going from supporting tens of thousands of simultaneous requests [41] to hundreds of thousands of simultaneous requests per machine [30]. However, with the breakdown of Dennard scaling for processor speeds [9], hardware architectures are forced to increase performance by placing multiple cores on a processor to exploit hardware-level parallelism. With the advent of parallel architectures, there is a *forced* shift in software development from the sequential to multi-threaded paradigm.

The ubiquity of commodity multi-core machines brings an increased emphasis on writing highly-concurrent software to take advantage of the added processing power [66]. Utilizing parallelism is particularly relevant to scaling web-application services, since modern servers need to handle traffic requests in the order of tens of thousands per second, which amounts to serving a few billion requests per day, and a few trillions per year [39]. Critically, the end of Moore's law is also looming on the horizon so that scaling of parallel architectures [35] is also going to see a decline. As a result, application developers will have to focus on more ways to optimize the use of concurrent software and parallel hardware, specifically exploiting caches and the I/O subsystem.

To exploit concurrent software and parallel hardware requires so-called *tuning knobs* in an application, with subsequent tuning adjustments and performance experiments over a variety of workloads to gauge effects. The bottom line is that there is no single solution for problems that works well across all workloads, so every application must compromise on some aspect of the workload to make others work better.

1.2 Hardware-Level Parallelism

There are several general architectures that support hardware parallelism. The *Central Processing Unit* (CPU) of a computer, typically referred to as the *processor*, contains discrete entities

such as memory caches, instruction decoders/pipelines, and units to perform arithmetic and logical functions. A computer that connects multiple such CPUs to a single shared memory via a memory bus is known as a *Symmetric Multi-Processor* (SMP). This architecture results in shared data that is logically in memory, but in reality replicated across many cache levels in each CPU. The complexity comes in ensuring a consistent view of the replicated data across all CPUs (cache coherency, memory model), while allowing maximal data duplication for performance reasons. By the 1990s, most OSs, compilers and programming languages supported SMPs for parallelism, working within a number of different caching and memory models.

As the number of transistors on a processor increased, the next step in parallelism is *Chip-level Multiprocessing* (CMP), where multiple CPUs reside on a single processor. Here, the term CPU becomes vague as there is nothing central or singular, but rather many processing units, so these CPUs are often called *cores*. However, in many designs, not all of the CPU is replicated for each core. Instead, a sufficient subset of a CPU's components are replicated per core, so each core works independently, but a subset of external components may be shared. For instance, a CMP processor might have separate Level-1 caches (data/instruction) for each core or share the L1 cache between pairs of cores, but share all Level-2/3 caches among the cores on a processor. In general, cache sharing helps reduce high chip-cost and reduce power consumption, while aiding cache-coherence issues.

A further increase in sharing is *Simultaneous Multithreading* (SMT), exploits instruction-level parallelism, where internal components of a core are shared, e.g., the instruction pipeline is shared by two or more hardware threads [21]. In the best case, multiple instructions simultaneously execute across replicated or idle hardware resources, where the hardware manages the replicated and non-shared units. An implementation of SMT is Intel Hyperthreading [16] or AMD Zen multithreading [76]. Correspondingly, as hardware evolved from a single processor architecture to finer-grained forms of parallelism, so must concurrent programming-models.

1.3 Concurrent Programming

Concurrent programming is notoriously difficult compared to sequential programming because of the complexities of subdividing a computation and coordinating the subdivisions [59, 67, 63]. There are numerous ways of dividing a computation among independent threads, synchronizing events among threads, and providing mutually-exclusive access to shared data. Concurrency is used to access hardware-level parallelism with the goal of reducing the real-time of a computation, increasing its responsiveness, and providing fault-tolerance. Web servers, reverse-proxies, forward-proxies, network servers, High Performance Computing (HPC), databases, and many more applications utilize concurrency to access parallelism. In this work, high-performance web servers are examined, which represent an interesting concurrent problem, covering a broad range of concurrency and parallel issues.

Concurrent programming provides synchronous and asynchronous programming models:

Synchronous model: has blocking calls, identical to a sequential call, where control does not return until the operation is complete. The calling thread may block in the called routine for an event (synchronization or mutual exclusion), versus just performing a computation. Event

delays are handled by active blocking (busy-waiting) or passive blocking (thread stops and is asynchronously restarted). For user-level threading, blocking switches to another user thread to preserve concurrency and possibly parallelism.

Asynchronous model: has nonblocking calls, where arguments are stored (in a buffer), and control returns before the computation is started; the arguments are subsequently processed concurrently by another module. This model is simple when no result is required from the asynchronous call, e.g., in the producer/consumer pattern, the producer generates a value, sends it, and continues immediately; a consumer processes the value but may block if no value is available. The model becomes complex when the asynchronous call (eventually) returns a result to the caller. Multiple techniques exist for retrieving the corresponding result for a given call, e.g., tickets, call-backs, and futures. Additional complexity occurs, when the caller is juggling multiple asynchronous computations from different modules. Knowing when to block for results and finding alternative work when the results are unavailable is difficult, resulting in code that is hard to read and extend. Furthermore, managing the buffered call-arguments and return values becomes a significant storage-management issue for the caller and callee.

Under-the-covers, both methods must coexist; the difference is the user experience provided by the model. In general, the synchronous model is often simpler to write and debug because it directly matches the sequential-programming paradigm. Whereas, the asynchronous model requires a programming paradigm-shift, where call/return is used in sequential code, and call-compute-retrieve is used in concurrent code.

1.4 User-Level Threading

The formulation of the original C10K problem [41] addressed the inability of servers to scale beyond ten thousand clients at a time. Solutions often adopted event-driven programming in the absence of scalable user-level threads [71], with hybrid approaches using events for management and threads for I/O. The modern version of the C10K problem is the C10M problem [30], which addresses the inability of servers to handle beyond a million clients at a time. To achieve this goal, solutions require programming models that exploit the range of parallel hardware available today. In the absence of scalable, light and efficient threading libraries, event-driven solutions are still adopted.

In the face of growing code complexity of event-driven and hybrid systems, and increasing hardware-level parallelism, there is a strong motivation to build lightweight and efficient runtime systems that support threaded programming. Such a system should be capable of implicitly capturing application state for simpler application development and providing fine-grained resource control while delivering concurrent execution on blocking operations. There are two types of threaded runtime-systems: kernel level and user level. Kernel-level runtimes have relatively higher overhead since the OS must provide fairness and memory protection at the kernel level. User-level runtime systems have a smaller overhead because switching between user-level tasks that share state does not involve the kernel. Thus, fine-grained user-level multithreading offers a promising path to achieve good performance while simplifying application development compared to hybrid event-driven approaches.

1.5 Methodology: Why Web-Servers?

The methodology in this thesis is to examine how an assortment of well-tuned control/task-parallel [33] web-servers behaves in the presence of blocking disk and network operations, with respect to different client loads, by looking at request throughput. The goal is to determine if the fundamental trade-offs in the two approaches to writing concurrent programs (synchronous and asynchronous) offer fundamentally differing performance, or if both models are equally competitive. In essence, the goal is to determine real factors affecting performance of concurrent applications.

In order to study the different approaches to writing highly concurrent applications, the choice of application-under-test becomes relevant. High-performance computing (HPC) applications are rejected because most are data-driven, *embarrassingly* parallel, and in memory [15], which stresses neither the concurrent software nor the I/O environment (network/disk). Complex control-parallel applications, like databases, are harder to setup and analyse due to their complexity and penchant for running their own operating system (OS) like environments. Basically, the web server is a good *Goldilocks* application (neither too big or too small) to study control (versus data) concurrency with respect to CPU and I/O parallelism, which exercises the entire hardware environment, allows reasonable experiments to be constructed, and has a simple enough code base to understand how to tune it and why it behaves as it does. Thus, the challenges in designing a high-performance web-server make it an interesting case study in concurrent applications.

A particular bias in this thesis is user-level threading in the context of web servers. From 2000 onwards, languages like Go [31], Erlang [22], Haskell [37], D [11], and μC++ [13, 12] have championed the M:N user-threading model, and many user-threading libraries have appeared [75, 60, 77], including putting green threads back into Java [61]. If user-level threading is going to be the new paradigm for concurrent programming, this work begins examining its effect on an archetypal concurrent application, the web server.

The following sections outline the background of web-server design and implementation needed for this work.

1.6 Web-Server Architectures

Historically, there are three main architectures for building web servers — event-driven [56], pipelined (stage-based) [18, 74], and thread-per-connection [2, 72]. Interestingly, all of these approaches require events and threads to handle potentially-blocking network/disk I/O [53, 62, 68]. Hence, all web servers are hybrid event/thread approaches, where the distinction is the prominence of events or threads.

Event-driven servers often use only a single thread to manage requests and the state of request service, but rely on asynchronous non-blocking I/O and callbacks to efficiently scale to tens of thousands of web connections on a single processor. Event-driven programming is touted for its minimal intrusion upon the system, since it adds a thin layer on top of the OS kernel and requires very low overhead for handling I/O-bound workloads. Event-driven code is especially challenging to maintain and extend, because of the non-linear control flow that requires careful

capturing of system state, which adds to the complexity of development. Large event-driven applications can make code difficult to read, state-machine diagrams complex, and debugging painful, as seen with `async`/`await` in JavaScript. Finally, high-performance event-driven web-servers still need threads to deals to blocking I/O operations, which is often hidden in a backend stage.

Pipeline servers divide the event-driven engine into multiple stages connected by queues of state, where each stage may have several threads and the final stage has multiple threads to deal with blocking I/O. The subdivided event stages are run in parallel, where synchronization of state changes is managed by the interconnecting atomic queues. If a stage is a bottleneck, more threads can be added and/or multiple copies created and connected to the up/down stream queues to increase performance. This approach maintains the sequential nature of the event programming at each stage. Feedback information can be generated and communicated via the interconnect queues or out-of-band queues to provide dynamic load-balancing by increasing/decreasing parallelism at any stage.

Thread-per-connection servers move the threading necessary for the backend I/O operations to each event, using the thread stack to hold the state of the event, and folds the event engine implicitly into the threading runtime and its nonblocking I/O mechanism. The event engine is now accessed indirectly by the sequential execution of each thread, and potentially blocking routine calls only block the user-level thread while the runtime event-engine continues executing other user threads and polling for I/O completions. The result is a simpler programming paradigm but the programmer may be restricted by the interface to access certain components of the threading event-engine.

Claims have been made about the performance benefits of each of the three architectures [33, 55, 71]. However, it is often pointed out that thread-per-connection web-servers do not scale to high request rates (10K-100K connections per second) because of threading costs. For instance, Pariag et al. [58] claim that overheads incurred by the thread-per-connection architecture make it difficult to match the performance of well implemented servers requiring substantially fewer threads to simultaneously service a large number of connections. As well, the documentation for the high-performance event-driven NGINX web-server states:

> The common way to design network applications is to assign a thread or process to each connection. This architecture is simple and easy to implement, but it does not scale when the application needs to handle thousands of simultaneous connections [50].

Similarly, nodeJS, a popular event-driven programming model, states that long-running operations can "tie up" threads and cause the server to run out of available threads and become unresponsive [78].

With demand for higher throughput on commodity machines, trends indicate that programmers have shied away from thread-based servers and have moved towards event-driven servers over the past decade. Figure 1.3 [48] demonstrates this trend. Over the past decade, Apache, the leading thread-per-connection web-server has fallen in popularity in favour of NGINX, an event-driven server. (Microsoft is thread-per-connection, Google is proprietary with no architecture

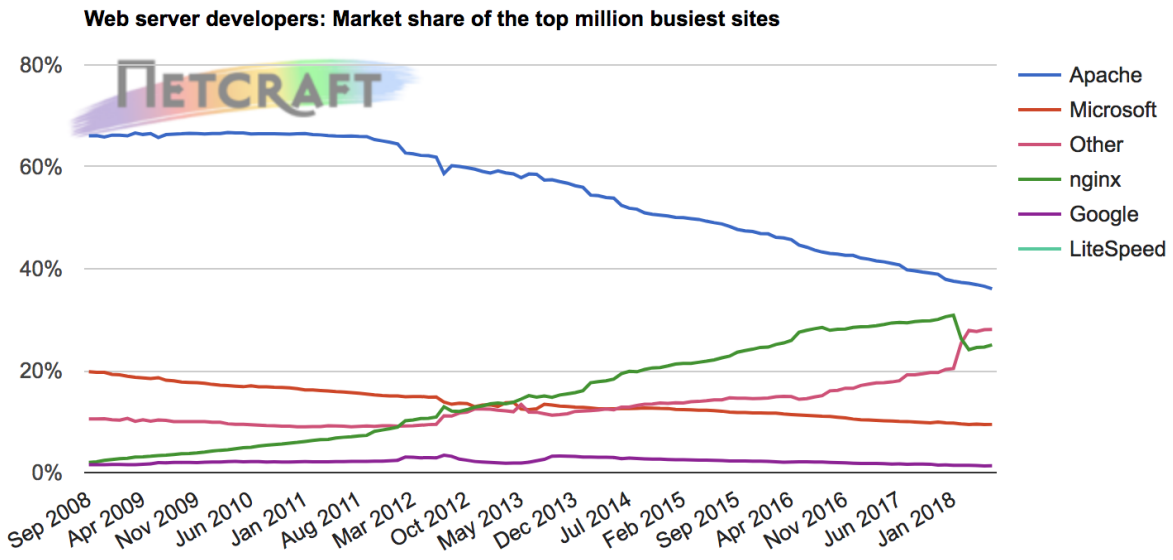


Figure 1.3: Web server usage trends over the last decade.

available [29], LiteSpeed is event driven (no line in graph.)) These trends and claims show that while thread-based models provide ease of programmability, performance is the primary factor driving design decisions.

Therefore, the fundamental thesis of this work centers around the following hypothesis:

- The thread-per-connection model simplifies building an inherently parallel application such as a web server by moving a significant amount of programming complexity into the threading subsystem.
- As a result, it is possible to program using synchronous APIs versus passing call-back routines or events for concurrent execution.
- However, thread-per-connection web servers using kernel threading versus user threading are less efficient.
- Therefore, there is a need for a lightweight, efficient threading library that can close the gap between ease of programmability and high performance.

1.7 Overprovisioning Perils

An additional aspect identified by this thesis is the problem of *unrealistic overprovisioning* in industry, where it is cheaper to scale out a system to obtain better performance and hide bad design, than pay the engineering costs to write highly tuned software that optimally utilizes the available hardware. Therefore, in addition to examining the two concurrent programming models, an additional hypothesis of this work is that constrained environments expose performance bugs that

otherwise might not be apparent in overprovisioned environments. Constrained environments can be synthetically engineered in one of two ways: by stressing the application-under-test using larger workloads, or by reducing the amount of hardware available to the application. The latter approach is chosen in this work, since it allows different aspects of an application to be stressed independently and observed. For instance, a web-server can be stressed by increasing the working set of files, the client request rate, or lowering the amount of memory, compute and network resources available to it. CPU-intensive applications can be stressed by reducing compute resources, and memory-intensive applications can be stressed by lowering the amount of memory.

The hypothesis implictly states that bottlenecks must come from the software and can be identified and examined in isolation. The problem is that this kind of environment does not reflect reality, where the hardware is always a bottleneck at some point due to its finite nature. Hence, adding hardware constraints can identify issues that can significantly differentiate software approaches, which would not otherwise be identified with essentially unlimited hardware.

This work asserts that hardware overprovisioning retards software development because there is less need to write optimized programs, i.e., even bad code runs "well enough". A general example of this problem occurred from 1970 to 1990, where increasing clock speed allowed sequential programs to remain static but performance increased dramatically. The new forced switch from sequential to parallel programming creates an opportunity to rethink, restructure, and rewrite many important applications. However, this opportunity is often thwarted by overprovisioning of parallel systems, because it is possible to double the number of cores, cache, and memory every 2-3 years. As with sequential overprovisioning, parallel overprovisioning masks poor concurrent design and implementation, making it difficult to determine if software optimizations are truly effective. For example, in the restricted context, eager web-servers can over accept connections, and grind to a halt or fail, while lazy web-servers allow connection requests to time out in favour of completing requests. Furthermore, any overprovisioning, other than to handle peak workload cases, results in unnecessary hardware cost, energy consumption, heat dissipation, and management [8, 43]. These costs have reached the point where there are global consequences in the form of a contributing factor to climate change. In fact, the environmental impact of data centers goes beyond just energy consumption. For instance, data center cooling systems release chlorofluorocarbon (CFC), a major contributing factor to global warming. Backup power systems to keep data centers running uninterrupted are typically run on fossil fuels, which have their own consequences. Finally, the reality is that there are many situations in the real world where software does run under restricted hardware constraints.

This observation begs a fundamental question: how much work can be accomplished on a fixed hardware configuration, i.e., how much work can be wrung out of a computer from highly optimized software? The global conjecture of this work is that good results from a constrained environment have a better chance of scaling well to increases along several hardware dimensions (e.g., CPUs, cache, I/O). However, both academia and industry suffer from the *bigger is better* syndrome. Therefore, in this thesis, the experimental environment has been restricted to 4 cores, 4GB or 2GB of RAM, 2 disk drives, and 8 $\times$ 1Gigabit Ethernet connections, which turns out to be an adequate but highly underprovisioned system by today's standards. Interestingly, all cloud services offer a VM of approximately this size, but the average sized VM is about 4 to 8 times larger [32]. Finally, this work does not address scaling, i.e., a web server that works well

in a constrained environment may or may not scale well in an over-provisioned environment. The conjecture is that many web servers work well in an over-provisioned environment, but the reverse stresses a server in ways unanticipated by its developers.

1.8 Contributions

The above theses and hypotheses are tested by examining web servers across the three architectures within a restricted hardware environment.

- two event-driven web-servers NGINX and μServer,
- one pipeline web-server WatPipe,
- one kernel-thread-per-connection web-server Apache,
- four user-level-per-connection web-servers goserver and gofasthttp [69] running on top of the Go programming language, YAWS running on top of Erlang programming language, and μKnot[1] running on top of the μC++ programming language.

During the work, the lessons learned from running performance experiments were fed back into μKnot and μC++, because of familiarity with this pairing. Trying to do similar changes to the other systems, i.e., changing the underlying language runtime and web-server internals, is beyond the scope of this thesis. It was challenging enough to modify one language/web-server.

The broad contributions of this thesis are:

- Testing web-servers under constrained conditions.
- A new, thread-per-connection web-server μKnot built atop the μC++ programming language. The implementation of this server includes a new concurrent metadata cache optimized for smaller memory footprint.
- Extending previous work comparing web-server architectures [58, 33] (WatPipe, μServer, knot) to include industry-grade (full-service) web-servers (Apache, NGINX) and user-level threaded servers (goserver, gofasthttp, μKnot, YAWS).

1.9 Outline

This thesis is organized as follows: Chapter 2 covers the background material required by the thesis, in addition to previous work done comparing web-server performance. Chapter 3 outlines the experimental environment used to compare a range of web-servers, and discusses the efforts involved tuning the servers individually for maximum performance. The web-servers chosen for

[1]The name is derived from the Capriccio thread-based web-server *knot*.

comparison span all three architectures: threaded (Apache, μKnot and GoFastHttp), event-driven (NGINX, μServer) and staged (WatPipe). Each server is tuned separately for light and heavy I/O cases, and subsequently profiled for analysis. Chapter 4 deals with boosting performance for μKnot. Chapter 5 contains the conclusion of the thesis, as well as future directions for the work.

Chapter 2

Background

Having sufficiently motivated the central thesis, the different web-server architectures can now be discussed using the synchronous and asynchronous concurrent-programming models. This chapter presents the web-server architectures that have evolved from the two concurrent programming models mentioned in Section 1.3, and describes efforts in previous work developing servers belonging to these architectures.

2.1 Web-Server Architectures

This section outlines the web-server architectures examined in this work.

2.1.1 Event-driven

This architecture is similar to the way an operating-system scheduler works. The server is in an event loop, dequeuing events from the queue, processing the event, and then taking the next event or waiting for new events to be pushed. Fundamentally, processing an event is accomplished by either statically registering an event handler for each specific kind of event, or dynamically registering a callback function at the point where the event is processed. The states of the client connections are managed explicitly using a finite state-machine. The *event engine* is a sequential program even though it is managing a number of disjoint operations [55]. Hence, the execution of the event engine is deterministic as long as no yielding operations are called in the event-handlers. However, this intertwined control-flow makes construction, maintenance, and debugging difficult. Furthermore, heavy utilization of the event engine can stall the single thread, i.e., the event engine cannot keep up with the request rate, which may require throttling the frontend stage and/or running multiple event engines. The main advantage of event-driven programming is the very low memory footprint because only the minimal state information is required for each event action.

However, both the front and back end of a web server must perform I/O, i.e., pull a request from a client and/or push the request result back to the client. These I/O operations can block

in the hardware and/or software for a large number of reasons. A blocking action performed by the event engine stops the web server for an unbounded amount of time, which is unacceptable. Hence, the event engine uses non-blocking I/O and manages these I/O completions itself, or spins-up threads for blocking operations and communicates with them to manage these I/O completions. In all cases, the event engine runs sequentially on a single thread. This approach requires complex I/O management and additional memory space used by the I/O thread, which must be factored into the event-engine memory-footprint.

The following sections present different kinds of event-driven web-servers architectures.

2.1.1.1 SPED

A common approach to an event-driven model is the *Single Process Event-Driven* (SPED) [57] architecture. SPED works by taking advantage of a secondary I/O event-engine and folding it into the primary web-server engine, i.e., incorporating the I/O front/back stages into the web-server event-engine. All client I/O (connections) are processed by the single web-server thread using non-blocking socket mode and only issuing system calls on those sockets that are ready for processing. The secondary event mechanism is provided by `select`, `poll`, `epoll`, or `kqueue` (see Section 4.1.4), which allow checking for sockets that are currently readable or writable. The server then uses this mechanism to process an event request without causing the server to block. However, the SPED architecture makes no provisions for I/O operations that always block, such as file I/O operations on certain versions of UNIX. Hence, this architecture is inadequate to deal with workloads that induced blocking due to heavy disk activity.

2.1.1.2 N-Copy

A natural extension to SPED is to run multiple copies, called an *N-copy* server [80]. Because each copy is independent (no shared information), when one copy blocks on I/O, another copy may be available to run, mitigating blocking I/O. However, if each copy listens for connections on a different port, there is a load balancing problem both for the clients and the server, i.e., the clients must be subdivided onto different ports as must the server copies.

2.1.1.3 Symmetric/Asymmetric

To eliminate explicit balancing, the N-Copy SPED processes can have one point of commonality, a listening socket, called a *Symmetric Multi-Process Event-Driven* (SYMPED) server [58]. The *Asymmetric Multi-Process Event-Driven* (AMPED) server, e.g., Flash Server [57], extends the backend of the SYMPED server to handle blocking I/O by using multiple operating-system processes or multiple kernel-threads within a process. In the former approach, no memory is shared (i.e., different address spaces), so communication is often done using low-cost pipes, while in the latter approach, memory is shared among threads so communication is performed through sharing data, Hence, the non-blocking I/O is folded into the event engine but blocking I/O is separated out into the blocking I/O stages, which requires additional overhead to coordinate between the event engine and the I/O stages.

2.1.2 Thread-Per-Connection

The thread based approach to building web servers associates each incoming client connection with a thread. I/O is then dealt with in a synchronous and blocking manner, which presents a straightforward sequential programming-model compared to the event-based approach. Threading also provides true concurrency in the system by isolating each client's request to a specific thread. Thread-based servers can be multi-process (one kernel thread per process) or multi-threaded (multiple kernel threads per process) threads, but there is always a 1:1 mapping of a thread to a connection.

The following sections present different kinds of thread-per-connection web-servers architectures.

2.1.2.1 Multi-Process

A well-known approach to designing UNIX servers is the *process-per-connection* model, which uses a new process, created via `fork`, for each connection [64], and `join` the process after the client request is complete. Hence, the process life time is the same as the connection life time. This approach isolates each new client request in its own address space. This model was used in the first HTTP server, CERN httpd [42]. Rather than dynamically fork/join for each client connection, some servers *prefork* a pool of worker processes when the server starts and manage this pool across the life of the server, possibly increasing/decreasing the pool size dynamically. In this scenario, each worker process blocks until a new connection arrives (using a thread-safe call such as `accept` in UNIX), processes the request, and blocks again. The workers can listen on the same or different ports depending on the load balancing approach. However, processes are inherently memory heavy and communicating among them is expensive.

2.1.2.2 Multi-Threaded

With the advent of standard threading libraries such as the Native POSIX Threading Library, new server architectures have emerged that take advantage of lighter-weight threads. A thread is lighter than a process in that multiple threads share an address space, and hence share global variables and state information. This capability makes it possible to implement features such as a shared cache for file access and cacheable responses for quick client responses. Another advantage of threads is their ease of creation and destruction, as well as the fact that they only need a new stack as opposed to a new address space. A disadvantage of threads is that the minimum stack-size usage is often larger than the storage required to manage events in an event-driven server, resulting in a larger memory footprint. Specifically, there is a minimum resident stack-storage accessed by the thread versus virtual stack-storage for unaccessed stack space. The general approach for thread-per-connection is quite similar to the process-per-connection approach, except for a significantly lower cost for communication.

2.1.3 Staged Event / Threaded

Welsh et al. [74] describe a hybrid approach to server design that employs both event-driven and thread-per-connection models to generate what is termed as the Staged Event-Driven Architecture (SEDA). SEDA consists of a network of event-driven stages connected by explicit queues (pipeline). Each SEDA stage uses a thread pool to process events entering that stage. Stages can either use their own thread pools, or share from a common pool of threads. The size of each stage's pool is governed by an application-specific resource controller, and threads are then used to perform the blocking I/O operations, thus lending itself naturally to concurrency by utilizing multiple CPUs. Stage based architecture is particularly useful for examining the load and performance of the server, since the size of the entry queue at each stage is a direct indicator of the server load. The separation of each stage also allows for a modular design – stages in the pipeline can be added or removed very easily. Pariag et al. [58] present WatPipe, another server that utilizes the pipeline architecture written in C++.

2.1.4 User-level Threading

User-level threading runtime systems extract more concurrency from threads by time-slicing a process thread into multiple user-level threads (N:1 threading). The underlying scheduling and management of user-level threads is performed by the language runtime-system, providing the programmer with a transparent API to the process thread. An extension of user-level threading is mapping user-threads across multi-threading in an address space (M:N threading).

Three concepts can form the building blocks of a user-level threading runtime: *user thread*-(UT)/*fibre*, *kernel threads* (KT), and *cluster*; Figure 2.1 [5] illustrates how UTs/fibres, KTs, and clusters relate to each other. UTs or fibres are the smallest execution contexts representing runnable tasks. Runtime systems differentiate between an UT and a fibre based on preemptive or non-preemptive scheduling. Hence, a UT is more concurrent because of implicit time-slicing between arbitrary instructions but more prone to concurrent errors, while a fibre is less concurrent because of explicit yield/blocks but safer because context switching only occurs at well defined points. Both UTs and fibres are executed collectively by a single process thread or multiple KTs. For multiple KTs, scheduling domains, called a *cluster*, provide pseudo-processes (shared memory) where the M:N model executes independently, possibly with different scheduling algorithms.

The runtime library provides a typical thread-based programming interface. An application can create and combine UTs/fibres, KT, and clusters to facilitate a desired execution layout. Each cluster manages a set of UTs/fibres by scheduling them for execution on the set of KTs associated with that cluster. UTs/fibres and KTs can migrate among clusters. Each cluster provides its own I/O multiplexer along with the data structures required to handle non-blocking I/O with the operating system. Note, the kernel I/O subsystem is global to a process, and all file descriptors are globally defined in a process and used by the cluster I/O multiplexer interacting with the operating system. Hence, it is possible for two clusters to use the same file descriptor and compete over access to this descriptor by multiple kernel threads.

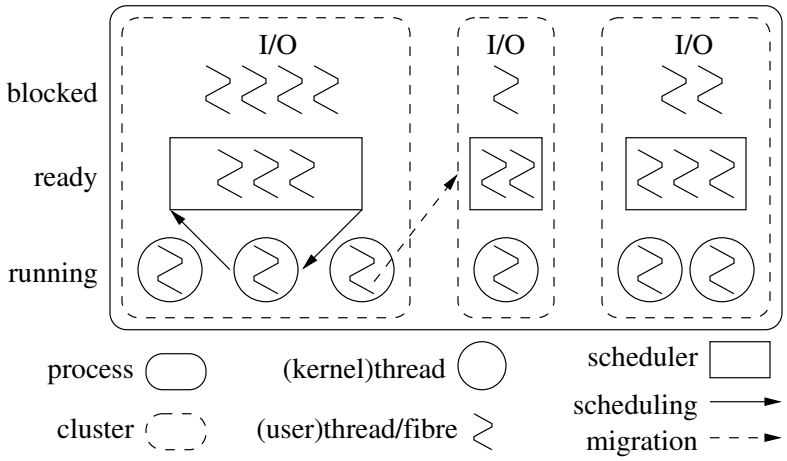


Figure 2.1: User-level runtime: process (address space) with 6 kernel threads and 3 clusters.

User-level threading runtimes provide support for network applications through some form of synchronous I/O interface, split into two categories:

1. frameworks that only support N:1 mapping (Capriccio [72], Windows Fibres [45], Facebook Folly [23], etc.) and
2. frameworks with M:N mappings(μC++, golang [31], C++ Boost [7], Erlang [22].

This work evaluates the efficacy of user-level threading with respect to high-performance web servers using the M:N systems, μC++, golang, and erlang.

2.1.4.1 μC++

μC++ is a branch of C++ that provides advanced control-flow including light-weight concurrency on shared memory multi-processors. μC++ provides new kind of classes to support concurrency: coroutines, which have independent execution states; tasks, which have their own threads; and monitors which allow for safe communication among tasks. μC++ is a translator that reads a program containing μC++ extensions and translates them to C++ statements and then uses a C++ compiler to compile and link them with μC++ concurrent runtime library. μC++ provides a M:N mapping of user-level threads (tasks) to kernel threads and clustering of user-level threads and kernel threads. The scheduling of user-level threads is performed using a round-robin preemptive-scheduler. Objects in μC++ communicate by sharing memory through routine calls, and mutual exclusion is implicit and limited in scope through the high-level concurrency constructs. μC++ provides static stack allocation where the minimum size of the stack is machine dependent, and can be as small as 256 bytes. Stack overflow checking can be turned on and is provided by a guard page at the end of the stack. I/O management is currently done through object oriented, non-blocking I/O using `select`.

2.1.4.2 golang

Go Programming Language (golang) [31] supports concurrency by providing user-level threads called goroutines. Goroutines communicate with one another and synchronize their execution through channels. Channels are adopted from Hoare's Communicating Sequential Processes (CSP) [36]. CSP is a formal language for concurrent systems that is based on message passing via channels. Go aims to provide a simple programming interface by removing complexity, and enabling safe concurrent programming. Go provides dynamic stack allocation by monitoring each function call and if an imminent stack overflow is detected, the runtime allocates a bigger stack and copies the old stack to the new one and updates all the pointers that point to the old stack. I/O management is currently done through non-blocking I/O using epoll. Go is a managed language, meaning it has garbage collection.

2.1.5 erlang

Erlang is a functional programming-language [22] supporting concurrency using user-level threads called processes that share no data. (The erlang name "process" matches with an OS process with respect to private memory.) Erlang processes communicate with one another using direct message passing (rather than indirectly via channels), where the message target is a named process in the send. Each process has a message queue where sent messages are buffered. A process receives messages from the buffer using pattern matching on the message type, which means messages may be serviced in non-FIFO order. Receive blocks the process if no message matches the pattern. Process IO requests are synchronous and the runtime system either performs non-blocking IO or dispatches an OS thread to perform blocking IO. Erlang is a managed language.

2.2 Workload and Content

The primary goal of all cloud-services platform today is scalability with reliability. Web-servers today face extremely large traffic volumes [39], and are required to deliver content at extremely fast rates that scale with this high demand. The content served by web-servers is of two kinds: static and dynamic.

Static content consists of files that do not change based on user input – the server simply publishes these files with each request. Static content has a few advantages:

- It is extremely fast. If a file is not expected to change often, the caching benefits from serving static content are enormous.
- Unlike dynamic content, which requires code to execute on the server or database connections to be made, static content is a more secure way of serving web content as there is no interactive element to hijack.

Dynamic content, on the other hand, is generated on the fly with each request. The document or content served only exists within the context of the request, although it can be cached as

well, depending on the pattern of consecutive requests. Dynamic content occurs in websites displaying customized content on a per-request basis. However, dynamic content suffers from a few disadvantages:

- The net throughput of the server is lower since dynamic content is more resource hungry.
- It is less secure than static content.

Although early generations of websites on the internet favoured static content, most websites today are a mix of static and dynamic content in an effort to combine the performance benefits of static content with the flexibility of dynamic content.

This work focuses on the simpler static workload because it only tests the web server with no dynamic (application) computation-component. Hence, only the web server and the underlying OS file-system are being used, so performance experiments do not measure other artifacts.

Chapter 3

Web-server Experiments

This chapter examines different web-server architectures by comparing the performance of different servers across two different workloads in two different hardware environments. The experimental setup is based on Harji [33], with slightly different parameters and additional web servers. The experimental workload and environment are discussed, followed by web-server tuning to get the best performance, and finally performance results.

To recap, web-servers architectures are as follows:

1. Event-Driven: This kind of web server uses an event loop to dispatch events that form the steps for servicing a client request. To prevent blocking I/O, multiple copies of the server are spun up to increase concurrency, called N-Copy, or a combination of kernel threads and asynchronous I/O operations are used, called SYMPED/AMPED. NGINX and μServer are examples of event-driven web-servers.

2. Pipelined: This kind of web server breaks the event loop into a series of concurrent stages connected by queues. Each concurrent stage is responsible for a particular step in the servicing of a client request, and within each step there is an event loop and possibly a thread pool for handling blocking operations. The state of each client's request flows through the pipeline via the connecting queues, until the last stage where multiple threads are often used to write date to the client to deal with blocking I/O. Pipelined servers may have a feedback mechanism between stages that allows dynamic tuning of particular stages depending on overload conditions at any stage, called SEDA. For example, if the connection processing stage is overloaded, it can be throttled, given more resources, or another connect stage added. WatPipe is example of a pipelined web-server *without feedback*.

3. Kernel-Thread-Per-Connection (KTPC): This type of architecture gives each client request its own event loop run by a separate *kernel thread*. Here the event loop is very simple because it only handles one request versus juggling thousands of requests. The complex event mechanism to handle the interactions among many requests in implicit in the non-blocking I/O library. However, spinning up large numbers of kernel threads ($>$100,000) can stress the OS. Apache is an example of a KTPC web-server.

4. User-Thread-Per-Connection (UTPC): This type of architecture gives each client request its own event loop run by a separate *user thread*. The behaviour is the same as for KTPC, except the UTs are multiplexed across a small number of KTs to reduce the stress and cost on the OS. goserver, gofasthttp, YAWS and μKnot are examples of a UTPC web-server.

The objective is to compare the performance of servers and understand underlying properties and limitations of server architectures. The servers are studied under two workloads: requesting a single fixed-sized file and variable-sized files chosen randomly from a power-law distribution (Zipf), where more small files are requested than large. The servers are studied in two environments: zero memory pressure, where the entire accessed fileset fits into memory (over provisioning), and medium memory pressure, where only 80% of the file set fits in memory forcing disk I/O to/from the file-system cache. Similar to prior work, Pariag et al. [58], each server is individually tuned for best performance so comparisons are as fair as possible. All tested web servers are open source (see Section 3.3).

The experiments are verified on two levels – correctness and fairness [33]. Correctness is ensured by checking that the file requested is received by the client by comparing the received file with a copy on the client. Because the comparison is expensive, correctness is only performed once (e.g., after adding/updating a server), and turned off when running throughput experiments, i.e., correctness is assumed to persist. Fairness is defined as the server processing all requests with equal priority. For instance, in a Zipf distribution of files and requests, it is possible to achieve higher throughput by throttling requests for larger files and giving priority to smaller files. Since smaller files stay in the server's file system cache and are requested more often (Zipf distribution), the server can process requests faster when not delayed by large files. To check for this behaviour, a limit is set on the number of times a request for a particular file can timeout. In concurrence with past experience, these checks determine if the server and environment are functioning correctly.

3.1 Benchmark

Each server is configured to process HTTP requests for static files. Processing a static HTTP occurs as follows:

1. The client opens a connection to the server and initiates a request for a file specified by a fully-qualified file-name.
2. The server listens for incoming connections, accepts the request, and reads the HTTP request from the socket.
3. The server *conceptually* locates the corresponding file on disk, reads it, and sends it to the client. In practice, the server may have simpler ways to combine and perform these steps.
4. If the underlying transport is persistent, multiple requests are serviced on the same connection, and then closed when there are no more requests from the client.

The server has a set of files with a size distribution generated by the old SPECWeb99[1] file generator [17]. For the experiments outlined in this thesis, there are 942 directories, each approximately 5MB in size, containing 40 files ranging in size from 102–921600 bytes, totalling approximately 4.7 GB. During an experiment a large subset of these files are requested by clients.

Each client has a log of file requests and uses `httperf` [47] to simulate thousands of concurrent users based on a specified request rate. The log file on each client is generated based on the server fileset and follows a Zipf distribution. Clients can request multiple files in a single HTTP session from the log file of requests. The log files model a person using a browser, with active and inactive periods that simulate a user activity and inactivity during a browser session [4].

In order to simulate realistic workloads, each client request needs to be serviced within a window. Some service providers even have Service Level Agreements (SLAs) [20] that guarantee a certain latency and throughput performance for clients. Therefore, if a request is not completed within a certain time window, it times out and the connection is closed. This behaviour models a user who waits for a finite amount of time for the web-page content to load. To simulate this behaviour in the experiments, a client request times out after 10 seconds. This value is generously bigger than any network or processing delay. The timeouts are enforced using the timeout parameter in `httperf`. Note, throughput is measured by the clients not the server, because the server may not see timeouts for a request that is never accepted. Hence, all graphs shows the client not the server experience during an experiment, which can be quite different.

Client latency is not measured because there is a latency bound of 10 seconds per request. Furthermore, client latency increases as the request rate increases and the server becomes busy. Once the server throughput peaks because some bottleneck is encountered, client latency plateaus as requests start timing out. Interestingly, `httperf` client-data shows the well-behaved servers self-throttle after peak, so more client requests are ignored as the request rate increases, independent of the request size, resulting in uniform timeouts. For a Zipf workload, clients perceive a slow throughput decline from peak as the server continues to service a large number of requests without network saturation because of the large number of small requests in the workload. For the fixed-size workload, clients perceive a sharp decline shortly after the peak as the network saturates quickly (50K packet $\times$ 8 bits $\times$ 20K request rate = 8 Gbps), so the number of timeouts grows quickly as the request rate increases.

To manipulate the environment to obtain zero and medium memory pressure for the tested web servers, the amount of memory (RAM) is toggled from 4GB to 2.4GB. For zero memory-pressure, a pre-experiment is run to load the file-system cache, so there are no reads; `vmstat` is run during each experiment to verify there are no *blocks-in* (reads). While the file set is 4.7GB, not all files are accessed, so 4GBs of RAM is sufficient, even when this amount is subdivided into only 3.4GB for file-system cache and 0.6GBs for the minimal OS space. Medium memory pressure occurs at 2.4GB RAM, representing an underprovisioned (realistic) environment. For disk I/O, a pre-experiment is run to load the file-system cache, but that does not prevent subsequent reads. At medium memory pressure, there is substantial I/O (noted in the `vmstat` output) because of file-request *churn*, i.e., the clients churn through the different file names at a sufficient rate resulting in file eviction from and reading files into the file-system cache.

[1] Both the SPECWeb99 and SPECWeb09 web sites indicate the benchmark is *retired*. However, retirement does not imply the benchmark is inadequate or wrong.

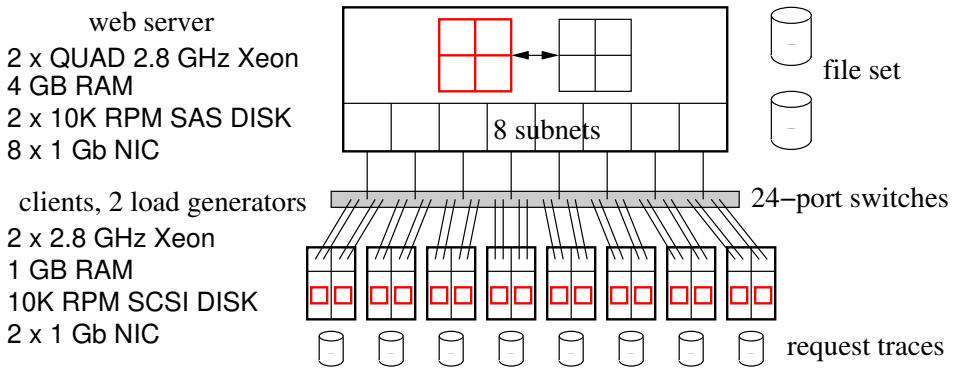


Figure 3.1: Experimental Setup

Where possible, the multiprocessor experiments have been optimized by pinning network interfaces to each CPU so that network interrupts are distributed equally among the CPUs. Process affinities also make best use of the cache.

3.2 Environment

Figure 3.1 shows the experimental environment consisting of eight client machines and a single server connected via switches. A client machine contains two 2.8 GHz Xeon CPUs, 1 GB of RAM, a 10,000 RPM SCSI disk and four one-gigabit Ethernet ports. Each client machine runs two copies of the workload generator on a separate CPU. The OS is 2.6.11.1 SMP Linux in 64-bit mode.[2] The server machine contains two quad-core E5440 2.83 GHz Xeon CPUs, 4 GB of RAM, two 10,000 RPM SAS disks and 10 one-gigabit Ethernet ports. Four ports are on-board, four are from a quad-port Intel PRO/1000 PT PCI-E card, and the remaining two are from a dual-port Intel PRO/1000 PT PCI-E card. The OS is 3.2.0-126 SMP Linux in 64-bit mode.[3] To achieve maximum performance, the server is configured with 4 cores on a single CPU, eliminating communication issues among CPUs. While the hardware is old, there are no new-hardware features needed to run the web servers more efficiently in this scaled-down environment. Note, this work does *not* imply its results scale up; the focus of this work is overloading within the same scale or scaling down with overloading. Scaling up with over provisioning can require different solutions; often scaling down solutions do not affect scaling up, but the reverse may not be true.

The clients, network interfaces, and switches have been sufficiently provisioned to ensure only the server is the bottleneck to test the web servers. Multiple gigabit Ethernet-interfaces per machine are organized into separate subnets, allowing for explicit load balancing of requests. (This approach is equivalent to subdividing a 10Gb Ethernet port into multiple channels.) Eight subnets are used to connect the server and client machines via multiple 24-port gigabit switches. Each client runs two copies of the load generator, with each copy using a different subnet to

[2] This OS kernel is very old. However, it does nothing but run `httperf`, and hence, there is no reason to update it and possibly perturb experiments.

[3] This OS kernel is recent (March 2017), but not new.

simulate multiple users sending requests to and getting responses from the web server. The subnets are distributed so the clients are equally spread over the eight interfaces available on the server. Hence, the clients and server communicate using fast, reliable network links. Based on a `netperf` experiment, the server achieved throughput of 7.5 Gbps, which is close to line-speed. The best web-server experiments are below this throughput, indicating there is ample network and bus headroom.

3.3 Servers

Throughput performance is compared using 8 servers spanning multiple architectures. Apache is a very popular, industry-grade, KTPC server using a variety of threading models to service concurrent connections. NGINX is a popular, industry-grade, event-driven server used for high performance applications. Goserver, gofasthttp and YAWS are medium-grade[4] UTPC servers built on programming-language lightweight threads. μServer is an academic event-driven server. WatPipe is an academic pipeline server. μKnot is an academic UPTC server. The runtime environment for the UPTC servers are: Go for goserver/gofasthttp, Erlang for YAWS and μC++ for μKnot. With the exception of goserver and gofasthttp, written in Go, and YAWS, written in Erlang, the other servers are written in C/C++.

To make architecture-specific observations about the experiments, a significant effort was made to ensure all servers are compared on an even footing. Therefore, each server is carefully tuned to ensure it performed as well as possible in the test environment. For all servers, any ancillary mechanisms, e.g., logging, are turned off to maximize performance and reduce footprint. As well, different OSs have socket-level options to control behaviour, such as `TCP_NODELAY`, `TCP_CORK`, and `SO_ACCEPTFILTER`, which a server may or may not have as a configuration parameter. When available, option `TCP_NODELAY` produced a small performance benefit, and `TCP_CORK` did not affect performance; `SO_ACCEPTFILTER` is unavailable on Linux, and hence was not tested.

3.4 Apache

This section looks at Apache 2.4.37, how it functions, and which configuration parameters can be adjusted to tune it for maximum performance. The UNIX[5] Apache HTTP Server [25] is a *full-service* KTPC web-server with three different threading models (contained in the MPM module). All three models are N-copy, with the latter two providing model extensions.

prefork is basic N-copy, where a pool of independent Apache processes is managed, with each process handling a connection. This original model provides concurrency on versions of UNIX *without* multiple KTs in a process, so there is no shared information for aspects like load balancing. The amount of concurrency is low because of the high cost of process management, often restricted by virtual memory.

[4]There servers are not as robust. All of them failed in a number of places during testing.

[5]Only the UNIX version of Apache is relevant to this work.

worker extends the prefork model by having each Apache process run multiple KTs, where each KT handles a connection, which significantly increases concurrency. The reason the KTs are partitioned is shared-process resource-limitations, such as file descriptors and the TCP stack, which can become bottlenecks with high KT access. Hence, Apache shards the KTs across the processes to reduce sharing of process-specific resources.

event extends the worker model by incorporating non-blocking I/O to reduce the number of KTs by multiplexing them among ready I/O operation rather than having each KT block for every I/O operation. Hence, the number of KTs is proportional to the number of active I/O operations rather than the number of connections, which is advantageous for persistent connections where a client makes multiple requests.

Apache usually runs as a background task, a daemon (UNIX) or a service (Windows), created at system bootup and runs permanently because it manages all web interaction on a system. For this work, Apache is simply started for each experiment.

3.4.1 Tuning

Apache is a complex software package designed to handle virtually all aspects of web-server interaction, with hundreds of configuration directives [26], both static and dynamic. Many unnecessary configuration directives were elided, e.g., mime, logging, etc., because they are not pertinent to this work. Apache 2.4.37 with APR 1.6.5 and ARP-util 1.6.1 are used, built with the static configuration *event* model because of the high request rates (tens of thousands per second) in the experiments. Within the event model, only the following four dynamic configuration parameters have an impact on Apache's performance in the experiments.

ServerLimit is the maximum number of Apache processes.

StartServers is the number of initially launched processes. Since the experiments start and maintain high request rates, this value is set to ServerLimit to prevent slow creation of processes at the start.

ThreadsPerChild is the maximum number of KTs per Apache process.

ThreadLimit/MaxRequestWorkers are set to ServerLimit $\times$ ThreadsPerChild because the experiments require more KTs than the default limits, which are set low to "avoid nasty effects caused by typos".

Hence, the only two direct tuning-parameters are ServerLimit and ThreadsPerChild, from which the others are derived. Tuning involves creating enough KTs to handle the high request rate but subdividing the KTs into processes to deal with limitations of having too many processes and too many KTs per process. Values like 32 processes with 8 KTs per process allow 256 KTs to handle requests.

3.4.2 2GB Tuning

Figure 3.2 shows a tuning graph generated by varying the ServerLimit and ThreadsPerChild parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 16K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. For all lines, there is a peak, and up to 3 data points are missing for large numbers of processes, e.g., 64 processes $\times$ 1024 KTs per process is 65536 KTs for the experiment, which causes Apache to generate errors so these results are excluded. The highest throughput is 1658 Mbps for 32 processes with 8 KTs per process. Therefore, the configuration of 32 $\times$ 8 is used for running the rest of the 2GB experiments at different request rates.

3.4.3 4GB Tuning

Figure 3.3 shows a tuning graph generated by varying the ServerLimit and ThreadsPerChild parameters with the environment configured at 4GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 20K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. For all lines, there is a peak, and 3 data points are missing for large numbers of processes, e.g., 64 processes $\times$ 1024 KTs per process is 65536 KTs for the experiment, which causes Apache to generate errors so the result is excluded. The highest throughput is 1684 Mbps for 64 processes with 64 KTs per process. Therefore, the configuration of 64 $\times$ 64 is used for running the rest of the experiments at different request rates.

3.4.4 Alternate Tuning

Similar tuning was done for the Apache worker-model (not shown) versus event-model, where 50 $\times$ 200 was selected for the ServerLimit and ThreadsPerChild parameters. Figure 3.4 compares the two models at 2GB, and Figure 3.5 compare the two models at 4GB. Notice there is no clear winner between the two models; there are places where one model is better across a certain duration and vice versa.

3.5 NGINX

NGINX 1.13.7 is an high-performance, *full-service*, event-driven, AMPED web-server (see Section 2.1.1.3). It can also serve as a reverse proxy and a load balancer [62]. Concurrent connections are handled using the complex event-driven architecture. The NGINX server runs a master process that performs privileged operations such as reading configuration files, binding to ports, and controlling worker processes. NGINX uses a disk-based cache for performance, and assigns a dedicated process to manage the cache. This process, known as the *cache manager*, is spun-off by the master process. Additionally, there can be many *worker processes*, each handling network connections, reading and writing disk files, and communicating with upstream servers, such as reverse proxies or databases.

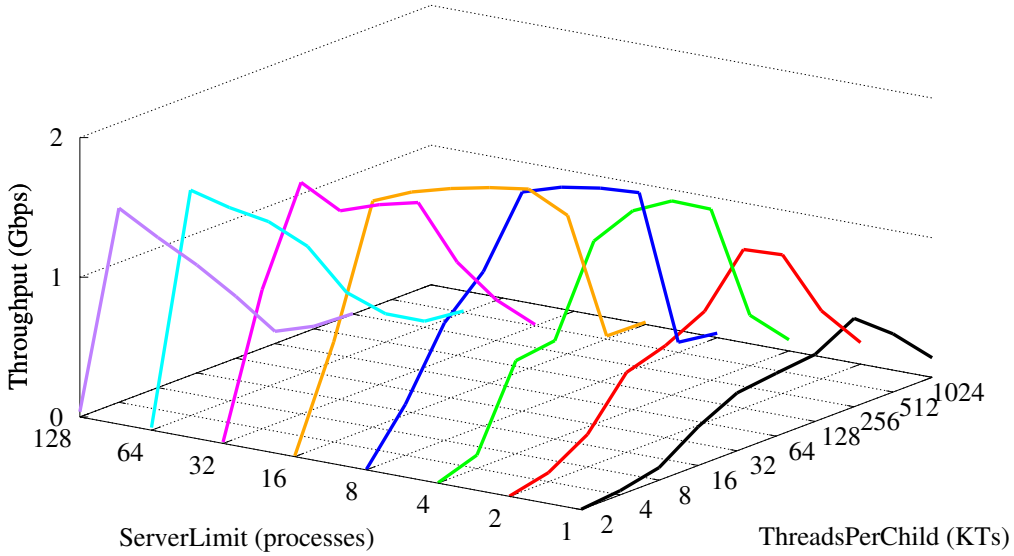


Figure 3.2: Apache Tuning: Event Model, 16K request rate, 2GB RAM

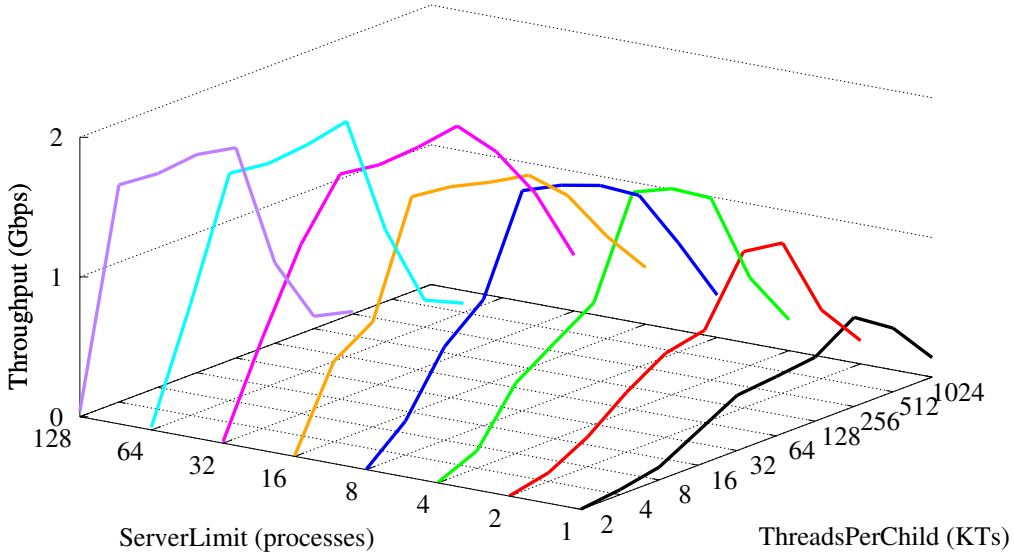


Figure 3.3: Apache Tuning: Event Model, 20K request rate, 4GB RAM

A worker is a single-threaded process, running independently of other workers. The worker process handles new incoming connections and processes them. Workers communicate using shared memory for shared cache data, session data, and other shared resources. Each worker assigns incoming connections to an HTTP state-machine. As in a typical event-driven architecture, the worker listens for events from the clients, and responds immediately without blocking. Memory use in NGINX is very conservative, because it does not spin up a new process or thread per connection, like Apache. All operations are asynchronous – implemented using event notifications, callback functions and fine-tuned timers.

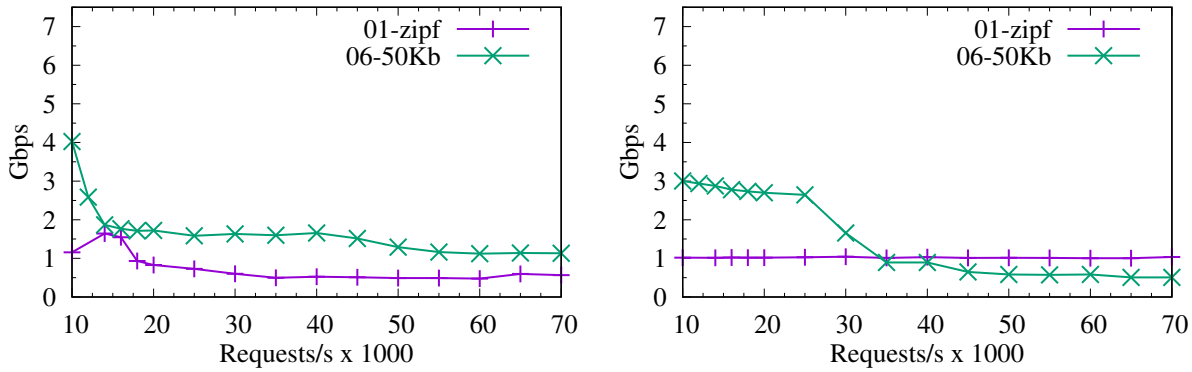


Figure 3.4: Apache Event (left) versus Worker (right) Models, 2GB

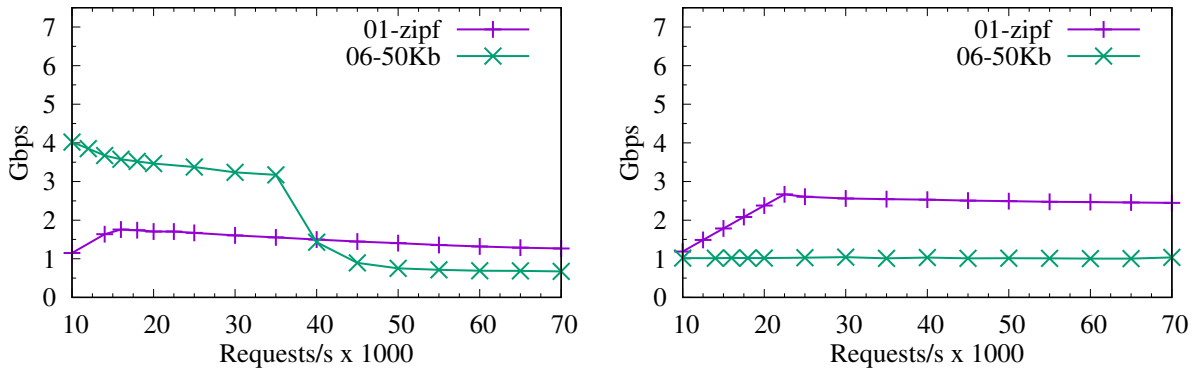


Figure 3.5: Apache Event (left) versus Worker (right) Models, 4GB

3.5.1 Tuning

A number of mechanisms and configuration file directives exist to mitigate disk I/O blocking scenarios for static content [52]. NGINX contains a self-managed server data-cache, implemented in the form of hierarchical data storage on a filesystem. This cache is part of the shared-memory segment accessible to all NGINX processes and necessary for dynamic-context workloads. For static-content, NGINX provides `sendfile` mode that relies on the file-system cache. Hence, it is possible to prevent copying a file into and out of the web server to obtain better performance and avoid blocking the worker process when a file is cached. This configuration option is an aspect of NGINX that makes it competitive in an under-provisioned environment. Since all the experiments are static content, server caching is disabled and `sendfile` is enabled.

NGINX recommends adjusting the following tuning parameters [51] for good performance:

worker processes : This option controls the number of worker processes (default 1). NGINX recommends a minimum of one worker process per CPU core, and increasing the value when there is significant disk I/O [51].

worker connections : This option is the number of connections each worker process can handle simultaneously (default 512). NGINX states the appropriate setting depends on the size of the server and the nature of the traffic and can be discovered through testing.

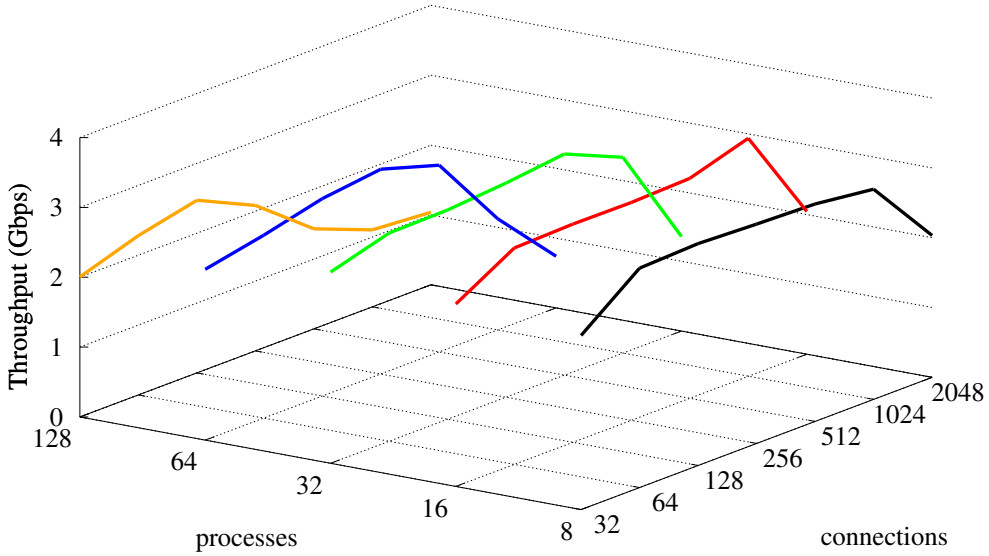


Figure 3.6: NGINX Tuning: 30K request rate, 2GB RAM

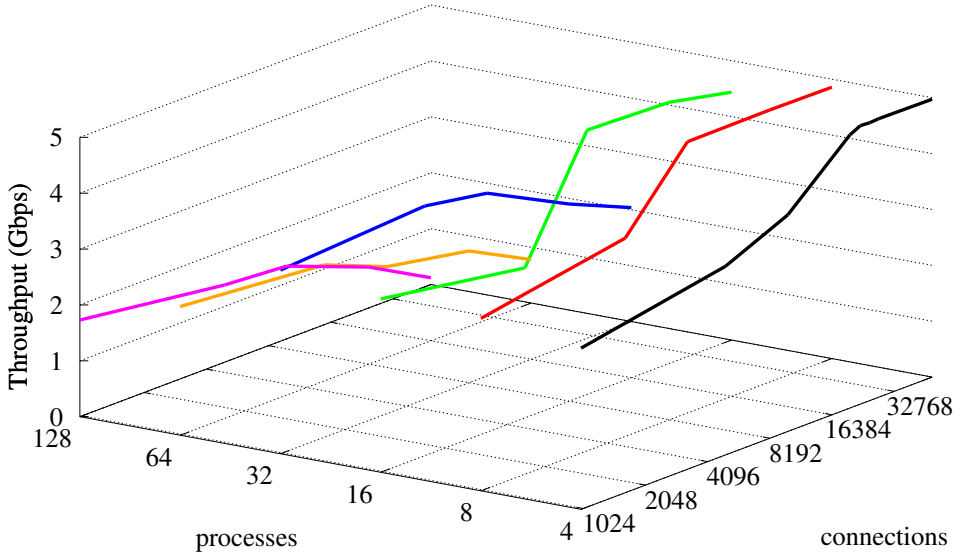


Figure 3.7: NGINX Tuning: 57.5K request rate, 4GB RAM

NGINX states parameters `keepalive_requests` and `keepalive_timeout` can have a significant affect. These parameters ensure a client connection is not dropped after each request, which speeds up persistent connects with multiple requests. However, the defaults for these parameters are 100 and 75 seconds, respectively, which are significantly above the maximum 6 persistent requests and 10 second client-timeout in the experiments. Experiments were run varying these parameters (not shown) to confirm the default values are adequate.

3.5.2 2GB Tuning

Figure 3.6 shows a tuning graph generated by varying worker processes and connection parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 30K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. Adding processes increasing throughput up until 16, and increasing connections helped to about 1024. The highest peak througput is 3407.8 Mbps for 16 processes with 1024 connections. For disk I/O, there is a need for more processes due to the blocking I/O. Therefore, the configuration of 16 kernel threads $\times$ 1024 connections is used for running the rest of the 2GB experiments at different request rates.

3.5.3 4GB Tuning

Figure 3.7 shows a tuning graph generated by varying worker processes and connection parameters with the environment configured at 4GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 57.5K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. Adding processes did not increase throughput, and increasing connections only helped for 4-16 processes. The highest peak througput is 4983.0 Mbps for 4 processes with 22.5K connections. Therefore, the configuration of 4 $\times$ 22.5K is used for running the rest of the 4GB experiments at different request rates.

3.6 goserver

`goserver` [28] is a *full-service* HTTP file-server written in the Go 1.10 programming language. It is a UTPC server – it creates a new *goroutine* for each incoming connection. The server is provided as a single-line golang standard-library call for implementing HTTP services:

```
http.ListenAndServe( ":"+strconv.Itoa(*port),
                     http.FileServer(http.Dir("/.../fileset")))
```

Additional code is added solely to vary the following tuning parameters.

3.6.1 Tuning

The following tuning parameters control the goserver and golang:

1. `GOMAXPROCS` and `SetMaxThreads` limit the number of KTs in the golang runtime. The `GOMAXPROCS` parameter limits the number of KTs executing simultaneously, not including threads blocked in system calls. The `SetMaxThreads` parameter limits the maximum number of KTs that can be created, including running and blocked threads. `SetMaxThreads` is set to a large value so it does not interfere with the creation of KTs.

2. `MaxConns` – is the maximum number of simultaneous connections. The goserver allows an unlimited number of connections by default (for the version being tested). At high request rates, throttling the number of simultaneous connections is necessary to prevent the acceptor thread from over connecting, which resulted in the server not processing accepted connection and high error rates by the clients or even server failure. (We believe the failure results from a cascade of allocations and garbage collections that ultimately resulting in a memory failure.) Hence, this parameter is tuned to a lower number than the higher peak request-rate to throttle accepts.

3.6.2 2GB Tuning

Figure 3.8 shows a tuning graph generated by varying the `GOMAXPROCS` (processes) and `MaxConns` parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 35K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. All experiments peak at 20K `MaxConns` and plateau thereafter. There is a slight performance gain from 4 to 8 processors, with no gains thereafter. The highest throughput is 2160.6 Mbps for 8 kernel threads with 30K connections. For disk I/O, there is a need for more KTs due to the blocking I/O. Therefore, the configuration of 8×30K is used for running the rest of the 2GB experiments at different request rates.

3.6.3 4GB Tuning

Figure 3.9 shows a tuning graph generated by varying the `GOMAXPROCS` (processes) and `MaxConns` parameters with the environment configured at 4GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 35K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. All experiments peak at 20K `MaxConns` and plateau thereafter. There is a slight performance gain from 4 to 8 processors, with no gains thereafter. The highest throughput is 2298.5 Mbps for 4 kernel threads with 40K connections. For in-memory (no disk I/O), only one KT per core is needed, one per processes. Therefore, the configuration of 4×40K is used for running the rest of the 4GB experiments at different request rates.

3.7 gofasthttp

Like goserver, gofasthttp [69] (github Jan 10 2018) is written in golang, is a *full-service* UTPC server, and leverages parts of the goserver library-package. However, gofasthttp restructures the goserver library [38] and is established as the faster server using benchmarking [70]. The primary difference is gofasthttp uses a preforked thread-pool (or goroutine-pool) versus creating and deleting threads for incoming connections. The worker-pool model is faster because the preforking (allocation, initialization and deletion) is performed once versus performing these operation continuously for each connection. At high request rates, there is a significant performance effect,

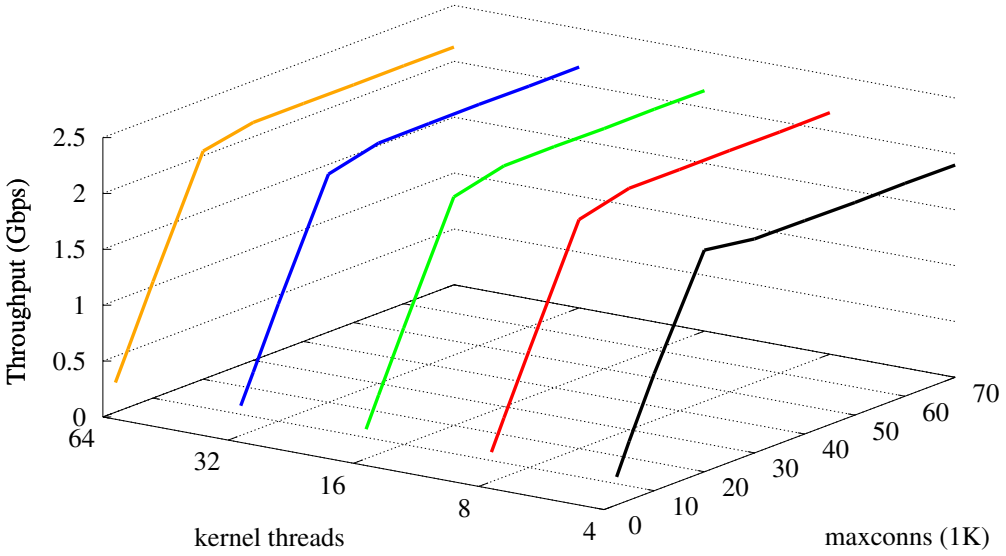


Figure 3.8: goserver Tuning: 35K request rate, 2GB RAM

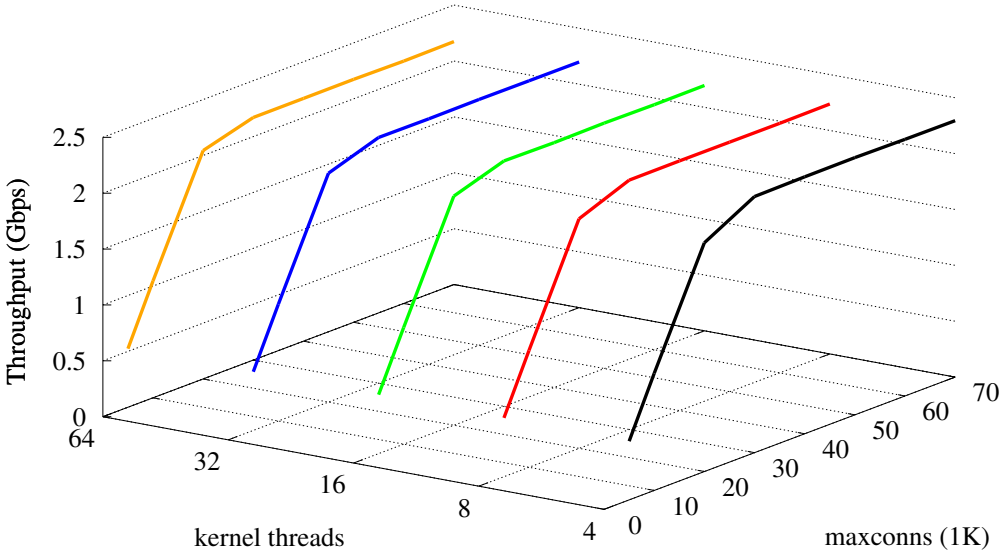


Figure 3.9: goserver Tuning: 35K request rate, 4GB RAM

especially because the garbage collector is rarely triggered due to the preallocations. Also, go-fasthttp request handlers maintain more context information than in the goserver library, which allows the handler to make fewer function calls and pointer redirects to obtain the information required to handle requests. Finally, the `ReduceMemoryUsage` option is turned on to aggressively reduce memory usage at the cost of higher CPU usage to limit memory in both the 4GB and 2GB experiments. Favouring space over speed works because web servers are typically I/O bound and CPU is not a bottleneck. When this option is enabled, reader/writer goroutines are buffered into their respective pools. A pool has elastic scalability, automatically creating new entries when the pool is empty and freeing entries back for garbage collection when the pool is to large. The pool also reduces garbage collection by retaining handles on the goroutines.

3.7.1 Tuning

The same tuning parameters as the goserver are used to tune gofasthttp (see Section 3.6.1).

3.7.2 2GB Tuning

Figure 3.10 shows a tuning graph generated by varying the `GOMAXPROCS` (kernel threads) and `MaxConns` parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 35K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. The results are unexpected. All experiments peak at 20K `MaxConns` (like goserver) but rather than plateau thereafter, the results fall shapely. The peak throughput at 20K MaxConns for processors 2,6–64 is between 1.5 and 2 Gbps. The anomaly is at 4 processors, where there is a spike of 3011.4 Mbps at 30K `MaxConns`. I have no explanation for this anomaly, but it is repeatable. Therefore, the configuration of 4 $\times$ 30K is used for running the rest of the 2GB experiments at different request rates.

3.7.3 4GB Tuning

Figure 3.11 shows a tuning graph generated by varying the `GOMAXPROCS` (kernel threads) and `MaxConns` parameters with the environment configured at 4GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 45K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. The results are identical in shape to the goserver 4GB (see Figure 3.9), but twice the throughput. The highest peak througput is 4858.0 Mbps for 4 kernel threads with 60K connections. For in-memory (no disk I/O), only one KT per core is needed, one per processes. Therefore, the configuration of 4 $\times$ 60K is used for running the rest of the 4GB experiments at different request rates.

3.8 μServer

μServer is an academic SPED server, where a single thread services multiple connections in various stages of processing using non-blocking I/O. The μServer has many configuration options, including using either `select`, `poll` or `epoll` as its event mechanism. It also supports zero-copy `sendfile` and caches HTTP headers and open file descriptors. μServer supports two different architectures to deal with blocking I/O:

- A master process creates a common listening port and then shares it with N independent μServer copies (N-copy). Except for the common listening port, no other information is shared, so no synchronization or mutual exclusion is required. When one SPED server blocks due to disk I/O, the OS context switches to another. The common listening socket means no additional port demultiplexing or load balancing is required.

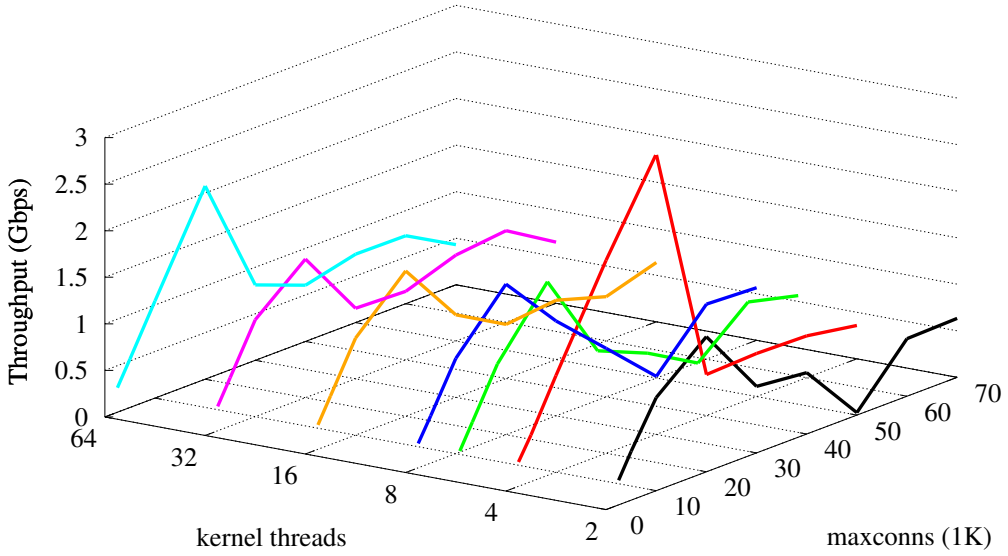


Figure 3.10: gofasthttp Tuning: 35K request rate, 2GB RAM

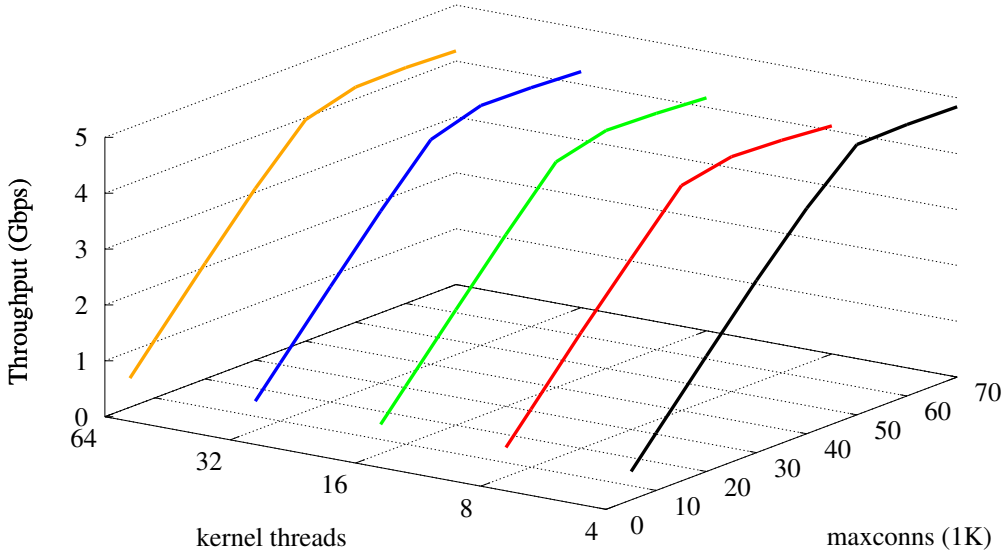


Figure 3.11: gofasthttp Tuning: 45K request rate, 4GB RAM

- The main drawback of the SYMPED architecture is duplication of HTTP headers and open-file cache. For example, 25,000 open files in 100 processes results in 2,500,000 file descriptors and associated HTTP headers. As well, the OS must support an equivalent number of open sockets and files. To mitigate duplication, the shared-SYMPED architecture `mmaps` a shared area among the processes for the file cache and mutual exclusion of the cache is handled by a single `futex` lock.

Since the other servers examined in this thesis all use some form of shared memory, the shared-SYMPED architecture is selected for a fair comparison.

3.8.1 Tuning

The following tuning parameters control μServer.

1. The number of copies (processes) created.
2. The maximum number of connections, which is used internally in μServer for fixed-sized tables to reduce dynamic allocation.

Tuning the shared-SYMPED μServer is difficult because I found a non-linearity as more KTs are added with respect to connections. More connections must be added as the KTs increase to prevent the server failing or providing unfair service to a subset of the clients. The unfairness results in the server dropping 30%-50% of these client requests, which results in apparent higher throughput because the server is doing less work. Rather than construct a non-linear formula for increasing connections as KTs increase (which is difficult to graph), I use the worst-case connections for the maximum 32 KTs. This choice means lower KTs have more connections than needed, which uses memory, and lowers their throughput. Hence, the final throughput results are slightly lower ($<$5%) than some specialized runs (not shown).

3.8.2 2GB Tuning

Figure 3.12 shows a tuning graph generated by varying processes with shared memory (which also implies the number of KTs) and connection parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 60K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. For a given number of kernel threads, the tunings all peak at 85K connections. The highest peak througput is 1766.7 Mbps for 8 kernel threads with 85K connections. For disk I/O, there is a need for more KTs due to the blocking I/O, but above 8 KTs, the extra connections needed cancelled out any benefit. Therefore, the configuration of 8 kernel threads $\times$ 85K connections is used for running the rest of the 2GB experiments at different request rates.

3.8.3 4GB Tuning

Figure 3.13 shows a tuning graph generated by varying processes with shared memory (which also implies the number of KTs) and connection parameters with the environment configured at 4GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 60K reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. For a given number of kernel threads, the tunings all peak at 90K-100K connections. The highest peak througput is 6786.7 Mbps for 4 kernel threads with 100K connections. For in-memory (no disk I/O), only one KT per core is needed, one per processes. Therefore, the configuration of 4 kernel threads $\times$ 100K connections is used for running the rest of the 4GB experiments at different request rates.

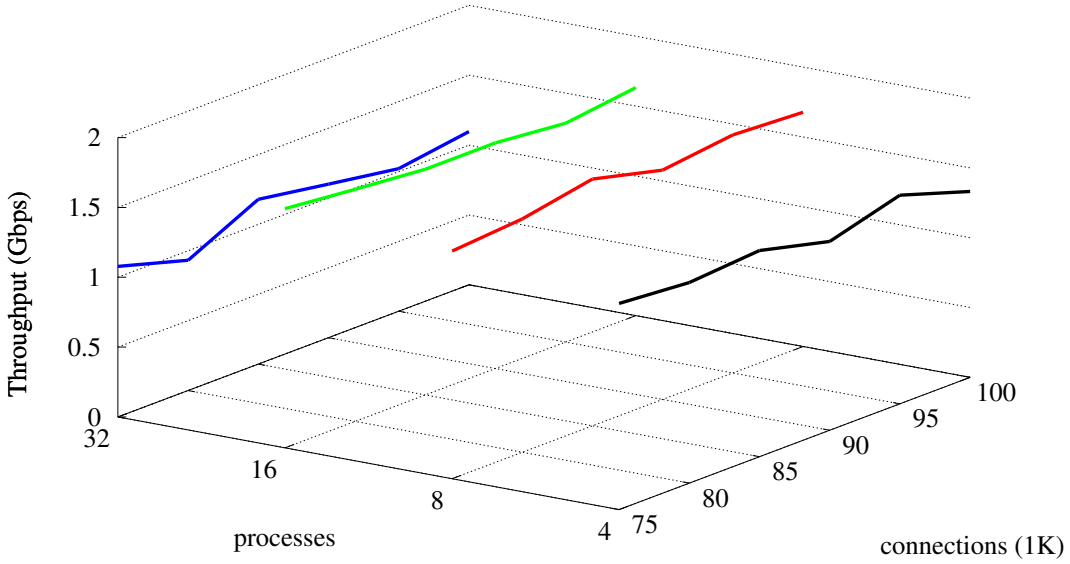


Figure 3.12: userver Tuning: 60K request rate, 2GB RAM

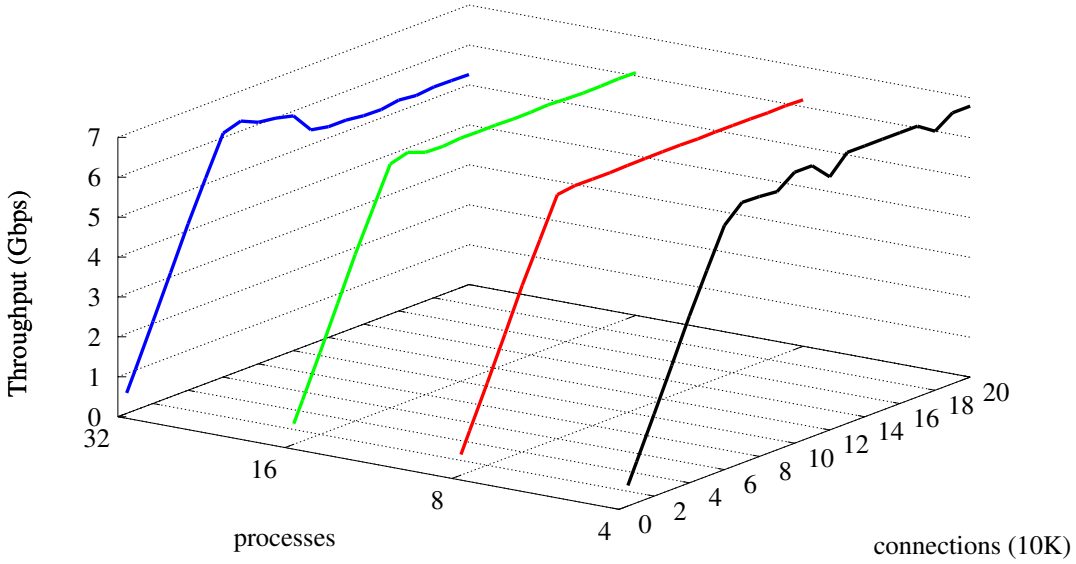


Figure 3.13: userver Tuning: 60K request rate, 4GB RAM

3.9 WatPipe

WatPipe is an academic pipelined server written in the C++ programming language, where each stage of the pipeline handles a portion of the HTTP request. WatPipe follows the SEDA design, but keeps overhead low by eliminating SEDA resource controllers with a short pipeline and a small number of threads at each stage. Keeping the number of threads small allows WatPipe to use kernel threads instead of user threads. Communication among stages is handled using explicit queues that are used to pass sockets. As an optimization, WatPipe batches events to minimize context switching. Like the other servers, WatPipe uses zero-copy `sendfile` and a metadata cache for HTTP responses. The server can use either `select` or `epoll` for events.

Specifically, the WatPipe implementation consists of 5 stages: Accept, Read Poll, Read, Write Poll and Write. The first 4 stages use one KT thread each, so there is no concurrency within a stage; stage 5 has a variable number of KT threads. Synchronization and mutual exclusion is required to communicate between stages and when accessing the open-file cache.

Stage 1 (Accept) accepts connections and passes accepted connections to stage 2.

Stage 2 (Read Poll) uses an event mechanism to handle the active connections that are readable and passes these events to stage 3.

Stage 3 (Read) reads the HTTP requests, parses them, and if not in the open-file cache, opens the file and updates the cache.

Stage 4 (Write Poll) uses an event mechanism to handle the connections that are writable and passes these events to stage 5.

Stage 5 (Write) uses a pool of KTs perform the actual writes, and these KTs may block on disk I/O. For non-blocking sendfile, a request may cycle at stage 5 until all bytes are written. On completion, the connection is passed back to stage 3 to handle persistent requests.

As well, WatPipe allows partitioning to distribute threads/IPs (see Section 4.1.1).

3.9.1 Tuning

The following tuning parameters control WatPipe.

1. The number of KTs used in Stage 5.
2. The maximum number of connections, which is used internally in WatPipe for fixed-sized tables to reduce dynamic allocation.

3.9.2 2GB Tuning

Figure 3.14 shows a tuning graph generated by varying kernel-thread and connection parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 57,500 reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. For a given number of kernel threads, the tunings peak at different numbers of connections: 4 at 50K, 8 at 60K, 16 at 60K, 32 at 30K, and 64 at 30K. The highest peak througput is 2176.1 Mbps for 64 kernel threads with 30K connections. For disk I/O, there is a need for more KTs due to the blocking I/O. Therefore, the configuration of 64 kernel threads $\times$ 30K connections is used for running the rest of the 2GB experiments at different request rates.

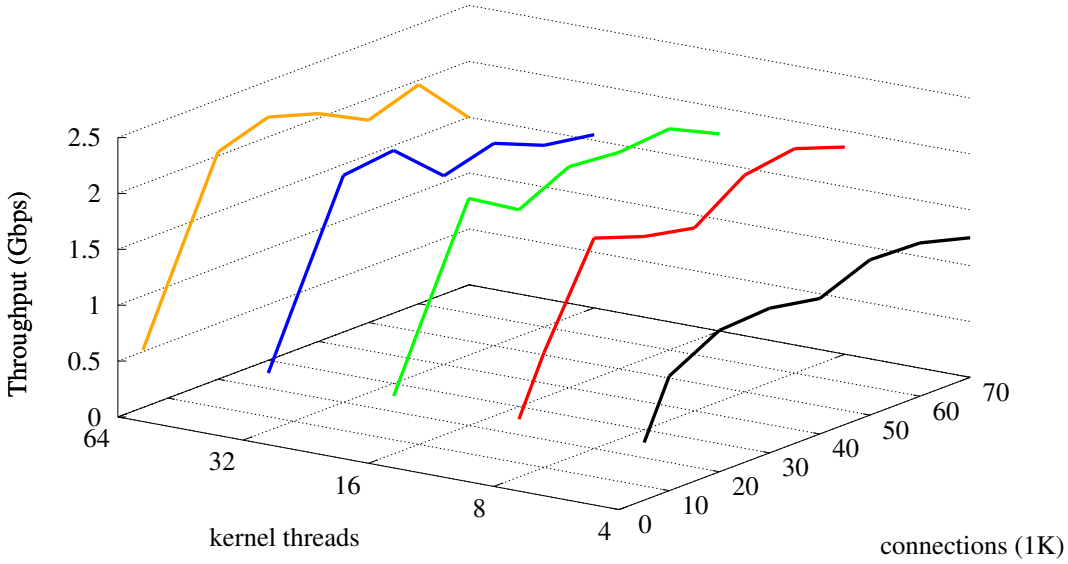


Figure 3.14: WatPipe Tuning: 57.5K request rate, 2GB RAM

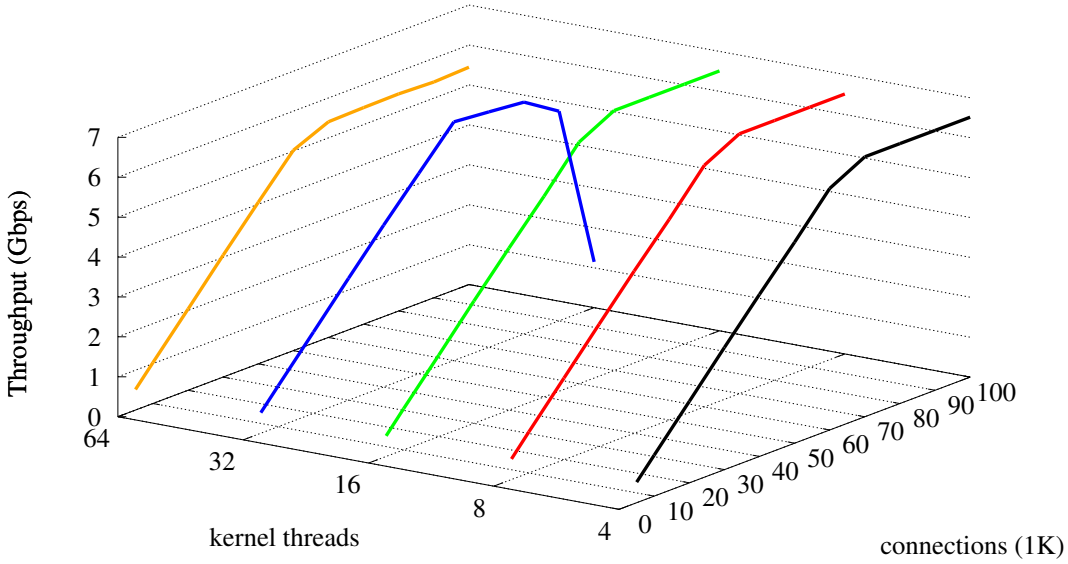


Figure 3.15: WatPipe Tuning: 57.5K request rate, 4GB RAM

3.9.3 4GB Tuning

Figure 3.15 shows a tuning graph generated by varying kernel-thread and connection parameters with the environment configured at 2GBs of memory. The graph shows the server throughput (Gbps) at a fixed client request of 57,500 reqs/sec for a Zipf distribution of retrieved files and the experiment duration is 300 seconds. For a given number of kernel threads, the tunings all peak at 70K connections. The highest peak througput is 6511.7 Mbps for 4 kernel threads with 70K connections. The dip for 32 KTs at 90K and 100K is repeatable and unexplainable. For in-memory (no disk I/O), only one KT per core is needed, one per partition. For in-memory (no disk I/O), only one KT per core is needed, one per partition. Therefore, the configuration of 4

kernel threads $\times$ 70K connections is used for running the rest of the 4GB experiments at different request rates.

3.10 YAWS

YAWS 2.0.6 is a *full-service* UTPC server written in the Erlang 21.0 programming language. Erlang [22] is a general-purpose, concurrent, functional programming-language with a garbage-collected runtime-system. It was originally designed to build soft real-time systems with high fault-tolerance for the telecommunication industry. YAWS also uses Erlang as its embedded language similar to PHP in Apache or Java in Tomcat for dynamic content.

3.10.1 Tuning

YAWS is a regular web server for delivering static content. By default YAWS caches static content in the server (application). Alternatively, there is the `large_file_sendfile = erlang | yaws | disable` option, which sets the `sendfile` method to send large files, where the default is `yaws`.

The caching behaviour is controlled by a number of global configuration directives.

- `max_num_cached_files = Integer` This directive controls the maximum number cached files. The default value is 400.
- `max_num_cached_bytes = Integer` This directive controls the total amount of memory for cached files. The default value is 1M bytes.
- `max_size_cached_file = Integer` This directive controls the maximum cached file-size. The default value is 8K bytes.

YAWS is the only server where all tuning attempts failed, resulting in poor to very poor performance. Hence, there are no tuning graphs for YAWS. While YAWS purports to be a static-content server, I was unable to force YAWS to use `sendfile` for all file transfers. As the YAWS documentation states, "sendfile method to send large files". Any files below this threshold are managed via YAWS own file cache, resulting in application copying. After extensive searching, no tuning knob was found to change the file size when `sendfile` is used, even after reading the Erlang source-code for YAWS.

An execution trace of YAWS during an experiment shows the lack of `sendfile` calls and the high cost of the application caching. The trace is generated using `strace -f -c` to obtain call counts to all applications processes:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 79.74 1748.809700        4691    372792     14080 futex
  8.19  179.668288        4386     40968           poll
  6.98  153.141348        1961     78089           epoll_wait
  1.22   26.854157           3  10023823           sched_yield
  0.81   17.672112         116    152958           epoll_ctl
  0.76   16.645428         125    133510     61349 recvfrom
  0.53   11.673210         151     77266         9 writev
  0.38    8.385872          68    124037     17718 getsockopt
  0.32    7.088329         173     40982           write
  0.22    4.909632         253     19409        30 sendfile
  ...
```

Surprisingly, 80% of the execution time is in calls to `futex`. It is also unusual to see calls to both `poll` and `epoll`. There are a large number of calls to `recvfrom` and `writev` potentially confirming the use of an application caching with copying. Finally, there are only 19,409 `sendfile` calls handling the larger files in the Zipf distribution.

An attempt was also made to adjust the caching parameters listed above. Even with all the values set to large amounts to preclude any restrictions:

```
max_num_cached_files = 50000
max_num_cached_bytes = 1000000000
max_size_cached_file = 50000000
```

there is little effect unless the values are dropped to low values. Essentially, there is a plateau where the caching parameters start working, after which there is little effect.

There is no reason for YAWS to have such poor performance, but I was unable to tune it for the workloads in the experiments given the available configuration parameters.

3.11 μKnot

μKnot is an academic web-server built to understand the fundamental performance costs for a web server using the UTPC architecture model and is built on μC++ with a M:N threading model supporting multi-core systems (see detailed discussion in Section 4.1). μKnot is *not* a full-service web-server (similar to WatPipe and μServer); it is a configurable bare-bones web-server to prevent conflating web-server features with pure runtime-performance. The configurations are:

1. maximum buckets in the cache hash-table
2. maximum cache entries
3. number of pre-spawned user-level threads
4. number of I/O kernel-level threads

5. number for CPU to bind I/O kernel threads
6. use IP address for client connections
7. use sendfile to write HTTP replies
8. cache HTTP-headers for sendfile
9. spawn user-level threads on demand
10. number of acceptor threads for auto-spawn
11. number of partitions to distribute threads/IPs
12. create a cache per cluster

The values selected for experiments are: sendfile, open-file cache, 20000 hash buckets, 22000 cache-entries, 4 partitions, CPU affinity, and a list of IP addresses for the separate client subnets. Since the number of files in the file-set is 22K, it is possible to set the number of hash buckets and cache entries to generate very short hash-chains and require no dynamic allocation for the cache entries. For auto spawn, up to N threads (where N is small) receive requests and each spawns another thread to process the request. Throttling the auto-spawners is an issue at high request rates, otherwise, they can flood the system. For pre-spawned threads, each thread accepts a request, processes it, and cycles back. Pre-spawned user-threads produced better performance than auto-spawning because creating and destroying user-threads is costly at high request rates, due to the contention on the shared heap for stack allocations.

Hence, the only two tuning parameter were number of user threads and I/O kernel-level threads. The number of user threads is proportional to the request rate. The number of I/O kernel-level threads is set to 4 for both in-memory and blocking I/O, where the system is partitioned into 4 (1 KT per partition) (see Section 4.1.1).

Other web servers use more KTs to handle blocking I/O, but these KTs are in a separate pool from those executing the other parts of the web server. The extra KTs in μC++ are in the pool used to execute threads from the ready queue, which does handle a blocking thread, but also puts stress on the runtime system to deal with idle KTs when there is no blocking I/O. In many cases only 1 KT is needed in each partition so additional KTs are put to sleep, otherwise there is unnecessary contention polling the ready queue. Now it is impossible to predict a blocking `sendfile` I/O, so the executing KT just blocks with no opportunity to restart one of the idle KTs to further execution. Having the idle KTs spin checking for work is to too expensive. Having them delay (short sleep) and then check is a game of picking the right delay length, and there is a cost in repeated delays. It is beyond the scope of this thesis to address this problem, so 4 KTs are used for disk I/O, which is why the results for the Zipf distribute are slightly lower than expected.

3.11.1 Tuning

As stated, many of the μKnot tuning parameters are fixed based on apriori knowledge.[6] Hence, the only tuning parameter for μKnot is the number of user threads.

[6] μServer and WatPipe are also able to use some aspects of the apriori knowledge.

3.11.2 2GB Tuning

Because there is only one tuning parameter, I took the opportunity to show more of the tuning process. Figure 3.16 includes a range of request rates, along with threads and throughput, versus a single tuning request rate. At 25K threads, performance peaked at 2065.0 Mbps for request rate 70K. Attempts to run experiments at 30K threads or higher lead to an assortment of OS failures, where the OS responded with error 12 "Cannot allocate memory for internal tables" on calls to `select` with the system hung requiring a reboot. At the failure, the maximum FD is around 45K–50K. Note, this failure is not a problem with limit on the number of FDs in the system because the same experiment is run with 4GB of RAM and no other change to the OS. Therefore, the configuration of 4 kernel threads $\times$ 25K user threads is used for running the rest of the 2GB experiments at different request rates.

3.11.3 4GB Tuning

Figure 3.17 includes a range of request rates, along with threads and throughput, versus a single tuning request rate. At 65K threads, performance peaked at 5784.6 Mbps for a request rate of 60K. Therefore, the configuration of 4 kernel threads $\times$ 65K user threads is used for running the rest of the 4GB experiments at different request rates.

3.12 Results

After tuning, the 8 web-servers are run using 4GB and 2GB of RAM and two workloads, Zipf and 50K single-file, across a series of request rates from 10K to 70K requests per second.

3.12.1 4GB

Figure 3.18 shows the results for the 4GB set of experiments, with request rate on the x-axis and server throughput in Gbps on the y-axis. The better the web server, the higher the throughput at a specific request rate. Each data point in the graph represents a 300 second experiment. Multiple runs for each data point were not performed because tuning experience showed performance was very stable ($\pm$1%). For each experiment, there is a small ramp up and down of about 15 seconds; hence, if an experiment is run longer, e.g., 1,200 seconds, the results are slightly better because the ramping period has less effect. There is no easy ways to ignore the ramp up/down because it varies with each run so it is hard to filter, and running longer experiments took too long. The shape of the 4GB Zipf (purple) and 50K (green) curves are *roughly* similar for all the web-servers.

The Zipf curves climb as the request rate increases until the curve peaks when the web server is saturated, after which clients begin to timeout because the server cannot respond fast enough due to some bottleneck, which is different for each web server. After the peak, throughput decreases slowly as the web server attempts to keep up with the increasing client load. With the

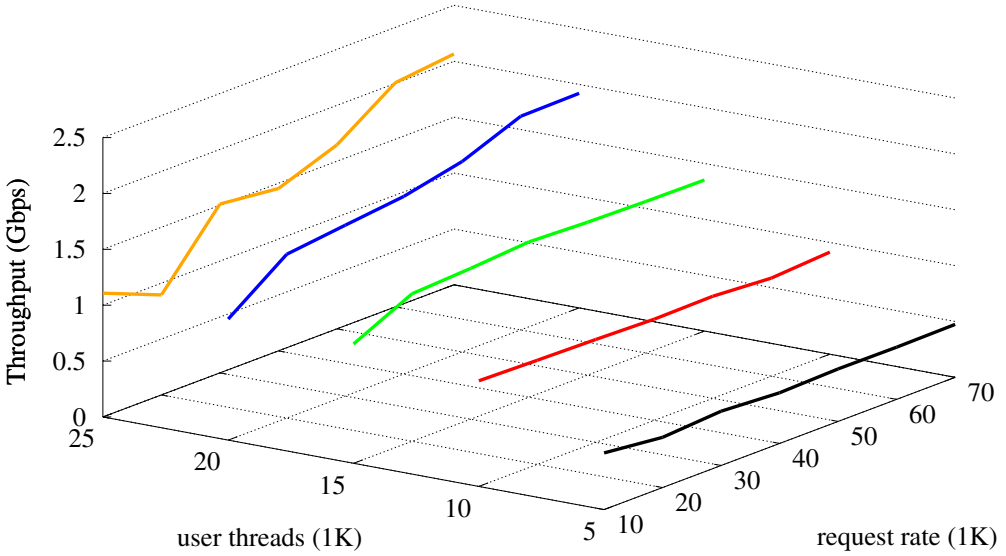


Figure 3.16: μKnot Tuning: 2GB RAM

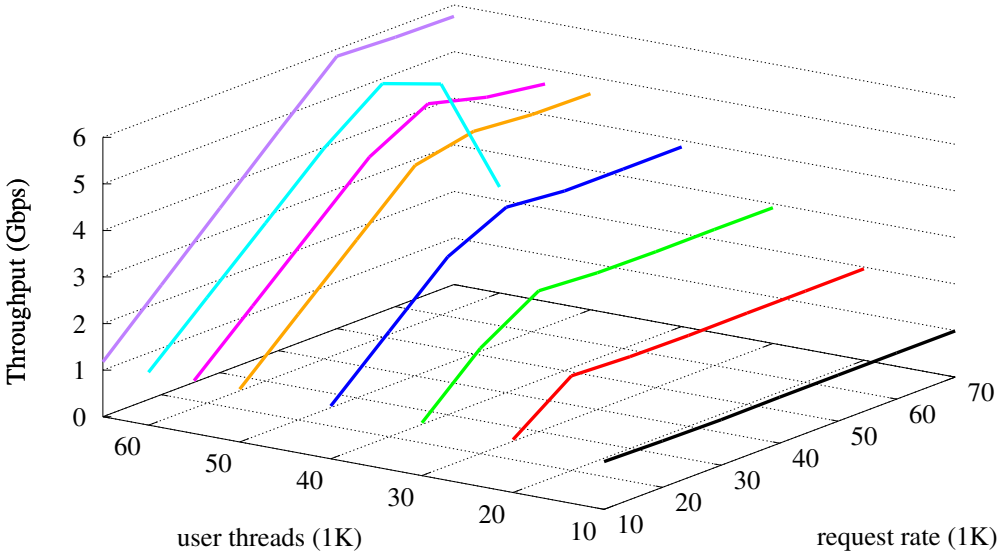


Figure 3.17: μKnot Tuning: 4GB RAM

exception of YAWS, this curve never collapses because all the servers were self-throttling, i.e., they service existing requests before accepting new ones. (Only the YAWS web-server collapses after the peak. However, the to unsuccessful tuning of YAWS means these results may not necessarily indiate YAWS' best-case performance.) Hence, more client requests are ignored after the server peak, independent of the request size, resulting in uniform timeouts. This observation is verified from the `httperf` client-data showing that all request sizes have a fairly equal percentage of timeouts. Since there are more small files, the total number of timeouts for these sizes increases significantly.

The difference in performance among the servers at 4GB for the Zipf workload is directly related to the usage of CPU and memory, which is observable from the `vmstat` output.

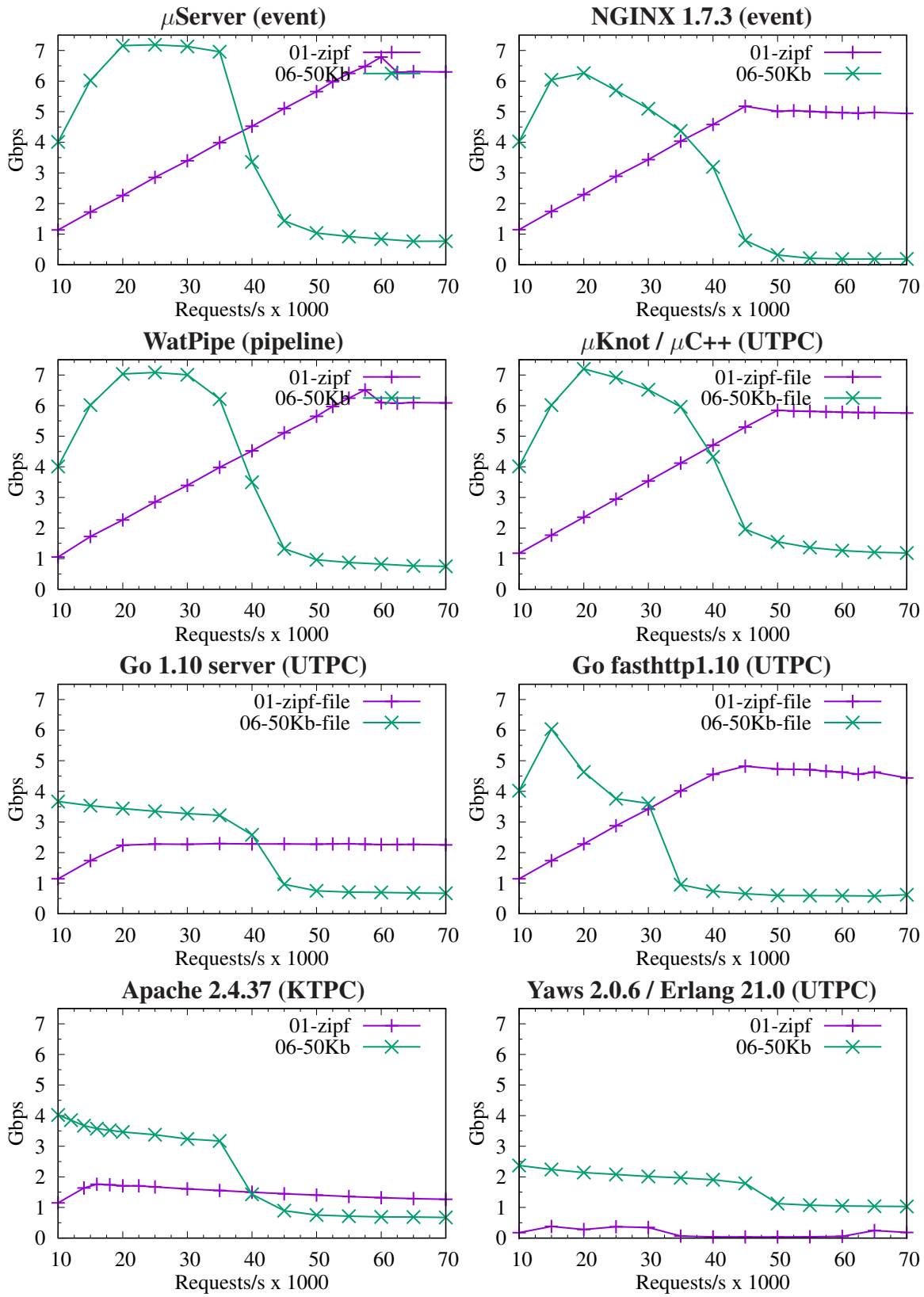


Figure 3.18: 4GB memory

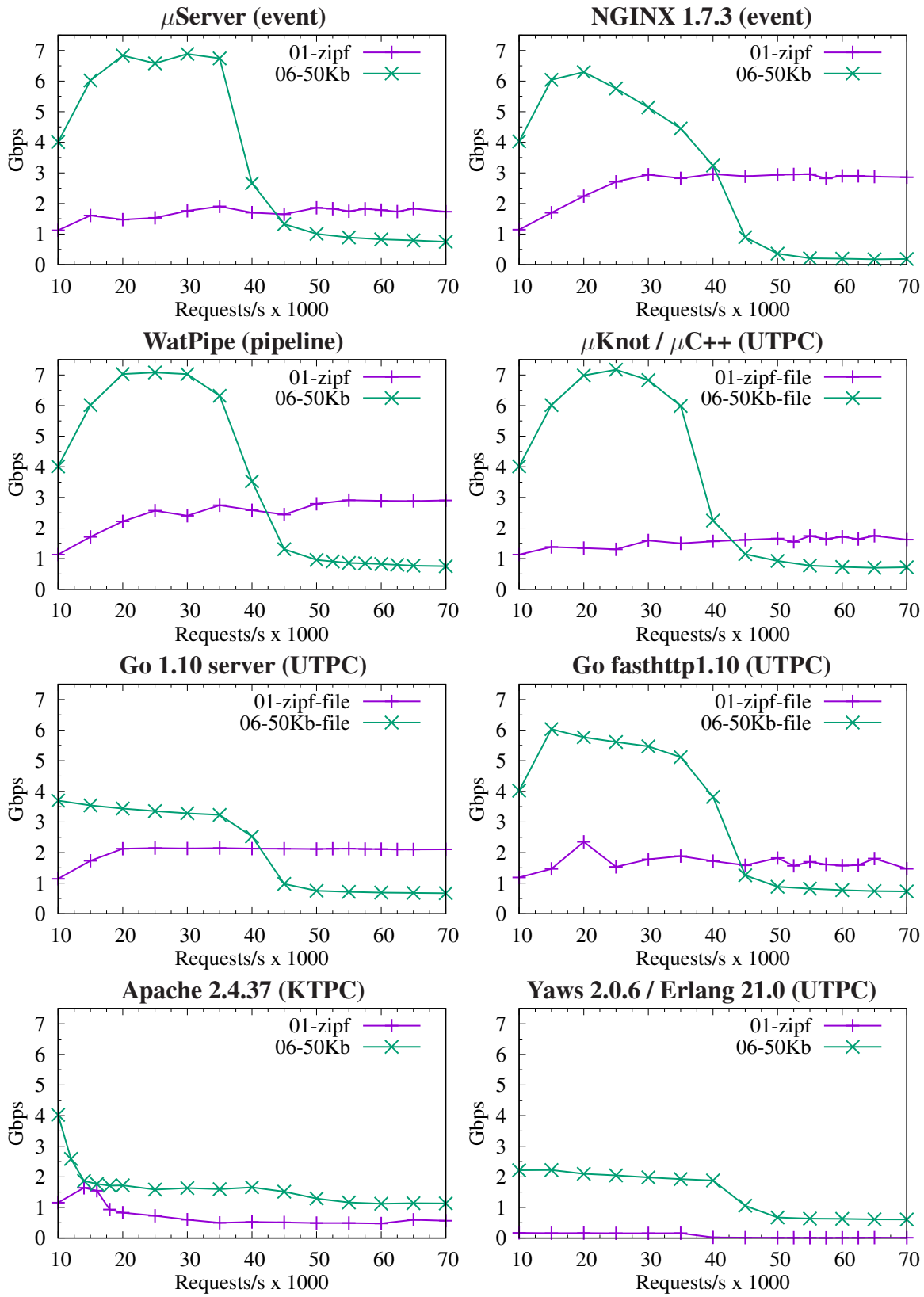


Figure 3.19: 2GB memory

server	reads (blocks-in)	CPU %
μServer	none	8
WatPipe	none	12
μKnot	none	5
NGINX	none	12
gofasthttp	200-500 intermittent	22
goserver	200-500 continuous	46
Apache	3000-4,000 continuous	19
YAWS	3000-7,000 intermittent	39

The servers are ranked from top to bottom, where top is better. The number of reads and CPU usage correlate with the observed server throughputs. It beyond the scope of this thesis to profile each server to determine the reasons for the additional resource usage. As well, the top web-servers cover all 3 basic server architectures: event, pipeline, and UTPC.

The 50K curves climb, peaks, and then collapses because the server and network saturate. At the 20K request-rate, there are 20K requests $\times$ 50K bytes $\times$ 8 bits = 8 Gbps, which is the maximum line speed to the server (see Figure 3.1). Throughput peaks at 20K with 7.2 Gbps for a short period until the network is overwhelmed and client throughput collapses due to timeouts. Note, server throughput remains close to line-speed during this experiment as it continues to service the requests it receives, which is verified by looking at the rolling statistics generated by some of the servers.

The difference in performance among the servers at 4GB for the single file is directly related to the usage of CPU, which is observable from the `vmstat` output. The only read is the single file, which is in the file-system cache.

server	reads (blocks-in)	CPU %
μServer	none	1
WatPipe	none	1
μKnot	none	1
NGINX	none	3
gofasthttp	none	2
goserver	none	6
Apache	none	3
YAWS	none	24

The servers are ranked from top to bottom, where top is better. The CPU usage correlates with the observed server throughputs. It beyond the scope of this thesis to profile each server to determine the reasons for the additional resource usage. Again, the top web-servers cover all 3 basic server architectures: event, pipeline, and UTPC.

3.12.2 2GB

Figure 3.19 shows the results for the 2GB set of experiments, with request rate on the x-axis and server throughput in Gbps on the y-axis. The structure of the 2GB experiments are identical to

the 4GB ones, modulo the difference in RAM memory. The shape of the 2GB Zipf (purple) and 50K (green) curves are *roughly* similar for all the web-servers, and bear some relationship to the 4GB curves.

I had numerous problems running experiments in the 2GB mode. For example, as request rates increase and servers are provisioned to handle a higher number of requests, the operating system would respond with out of memory, or hang and require rebooting. Furthermore, when results are obtained, they differ significantly from Harji's work for the overlapping servers. For example, the Zipf results are half of the Harji results, 2.5 Gbps versus 5 Gbps (see [33, Figure 4.4]). Extensive time was spent adjusting parameters in the servers and the OS with no changes. The conclusion is that the newer OS has a *regression issue* related to the file-system cache and disk I/O (see Section 4.2). This conclusion is not arbitrary because exactly the same server code from Harji's work is being used for WatPipe and μServer, and those servers produced twice the throughput on the same hardware. Therefore, it is hard to draw conclusions about these results, except that this kind of performance is occurring on current Linux systems. It is beyond the scope of this thesis to track down the regression.

The 2GB Zipf curves should match the 4GB curves in shape for the same reasons, with the different being lower throughput because there is disk I/O. However, in the 2GB configuration, the Zipf curves do not match the 4GB shape. Instead, they plateau quickly or continue to rise over the entire request-rate range. Furthermore, the client error-rate (dropped connections) is high across all the request rates, when there should be zero errors for the lower request rates (as in the 4GB experiments). Again, a significant effort was made to tune all the web servers in this environment with little success across all servers. Furthermore, this anomalous behaviour appears in all the web servers, pointing to a systemic problem in the OS.

As for 4GB, the `vmstat` results are presented at 2GB for the Zipf workload.

server	reads (blocks-in)	CPU %
NGINX	5,000-14,000	9
WatPipe	7,000-13,000	6
μServer	7,000-15,000	2
goserver	14,000-19,000 continuous	48
μKnot	5,000-8,000	2
gofasthttp	6,000-20,000 intermittent	2
Apache	4000-25,000 continuous	13
YAWS	1500-2,000 intermittent	35

The servers are ranked from top to bottom, where top is better. Due to the OS problem, it is difficult to draw conclusions about the reason for performance differences, except more reads slows performance. gofasthttp results for Zipf were very peculiar, as the experiment started with lots of the reads, but halfway through the experiment, the reads dropped to 0, suggesting some dynamic tuning within the server. The top web-servers cover all 3 basic server architectures: event, pipeline, and UTPC.

The 2GB 50K-curves should also match the 4GB curves in shape for the same reasons, with *no* difference in throughput because there is no I/O to read the single 50K file, which is in the

files system cache. However, for several web servers, the results are consistently different. Harji did not run a 50K single-file experiment, so there could be a bug in the μServer program.

As for 4GB, the `vmstat` results are presented at 2GB for the single-file workload.

server	reads (blocks-in)	CPU %
WatPipe	none	5
μKnot	none	5
μServer	none	4
NGINX	none	12
gofasthttp	none	15
goserver	none	31
YAWS	none	43
Apache	none	10

The servers are ranked from top to bottom, where top is better. Due to the OS problem, it is difficult to draw conclusions about the reason for performance differences, except more CPU slows performance (except Apache). As well, the top web-servers cover all 3 basic server architectures: event, pipeline, and UTPC.

Chapter 4

Performance Enhancements

Crucial to the performance of a static request is the content transfer from disk, because of the mechanical aspects of a disk drive. While solid-state disks (SSD) improve performance significantly, their cost per byte is still uncompetitive for large amounts of data, so spinning disk drive will continue for the foreseeable future. Hence, disk-drive I/O quickly slows the performance of any web-server once the request rate is high and the files are large.

To mitigate I/O, the OS typically runs an internal file-system cache, so actively used files stay in memory (similar to paging but for files rather than disk blocks). The file-system cache varies in size depending on the available memory and the memory required by applications running on the system. The OS monitors and dynamically adjusts the file-system cache to balance the working sets of application memory. The file-system cache is vital for a web-server – by caching a working set of the files in the file-system cache, the amount of disk I/O performed by the server is greatly reduced. In this case, transferring a file across the network can be a straight write from kernel memory to the socket. Because the file-system cache is managed by the OS, with only a few indirect kernel "tuning knobs", this crucial capability is beyond direct control of the web-server developer. The only indirect tool available to a developer is to keep the web-server footprint as small as possible, which correspondingly provides more memory for the file-system cache.

Part of I/O is transferring data from the file-system cache to/from an application. A web-server developer may be able to prevent double copying of file content from the disk (cache) into the web-server and than back out to the network, by using an OS mechanism, e.g., `sendfile`, to copy from the file-system cache directly to the network. Eliminating double copy provides a significant performance benefit for a web server.

Another important factor when dealing with high request rates is blocking I/O. If an I/O operation blocks the KT making the system call, the web server now has less concurrency to accept and process requests (down to the point where the web server stops execution until an I/O operation completes and restarts a KT). Therefore, strategies for dealing with blocking I/O are a critical part of any high-performance web-server.

The majority of blocking I/O is either network or disk. Historically, network I/O was slower than disk I/O, so non-blocking network I/O developed first in early UNIX, and non-blocking disk

I/O was ignored; hence, there is little or no support for non-blocking file I/O. However, performance between network and disk is now reversing, where network performance has significantly faster transfer rates than disk I/O, especially if the disk access-pattern is random. While asynchronous disk-I/O mechanisms, such as the POSIX asynchronous I/O interface (AIO), do exist to handle blocking disk I/O, they are fragile and lack support for performance-critical system-calls like `sendfile`, which are essential for removing double copy.

To mitigate blocking I/O, web-servers simulate asynchronous I/O using multiple KTs each performing a blocking I/O operation; however, this simulation is less efficient than a native asynchronous API due to additional threading overheads, especially the additional space required by the thread stacks, which steals from the file-system cache. Note, these threads do not transform blocking I/O into non-blocking – they only provide an alternate execution path so the server KT continues while the I/O KT handles the blocking operation. This approach relies on the OS context switching from the blocking KT back to the server KT. Hence, highly concurrent web-servers need to maintain a fine balance between the number of threads spun up to handle blocking I/O and the memory overhead associated with these threads.

UNIX-style OSs do provide mechanisms for non-blocking network I/O (see list in Section 2.1.1.1); The basic approach is to set each *appropriate* I/O file-descriptor (FD) to non-blocking mode. In this mode, when a create/transmit operations is performed on the FD, it returns either operation complete (ignoring errors) or would-block when some aspect needed by the operation is busy (e.g., socket buffers are full). It is then the application's responsibility to detect when the blocking condition has ended and then restart the operation (see Section 4.1.4).

To further eliminate I/O, a web-server can manage an explicit open-file cache, separate from the implicit file-system cache. The purpose of the open-file cache is to reduce system calls to the file system, which are expensive and perturb the disk when there is I/O to service these requests. For each client static-request, the server must `stat` the requested file for existence, and if it exists, obtain its size, because the HTTP response requires the file size in the header. While I-nodes containing file existence/size maybe in the file-system cache, they occupy valuable cache space and have to be accessed across the kernel boundary. The tradeoff is space versus time, i.e., the data-structure space/probe for the open-file cache, which steals from the file-system cache, and space/lookup for the I-nodes/`stat` I/O operation on each request.

In detail, the open-file cache is a relatively small hash table, where the key is the file name and the data is the file size. The implementation goal is to provide a good hash function for long file-names used by a concurrent hash-table shared by the requester threads. The hash table must also support evictions when there is I/O. In general, it is best to optimize for space rather than speed because the unused space goes directly back to the file-system cache. Any performance gain is not from the quality of the hash table, but rather reducing file `stat`ing. However, contention on the hash table can cause secondary problems for the requester threads. In general, most file-name accesses are reads because evictions occur infrequently or not at all for in-memory workloads. Hence, selecting a locking technique that optimizes reads over writes is normally sufficient.

4.1 μKnot

μKnot was built to understand the fundamental performance costs for a web server using the UTPC architecture model on multi-core systems. While its namesake Knot [72] was built for the same purpose, it was built on top of Capriccio with a N:1 threading model, so there is no access to parallelism. Whereas, μKnot is built on μC++ with a M:N threading model, so there is access to parallelism on a multi-core computer. μKnot is *not* a full-service web-server (similar to WatPipe and μServer); it is a configurable bare-bones web-server to prevent conflating web-server features with pure runtime-performance.

μKnot's configurable components leverage the design results from Harji's thesis:

1. Web-server partitioning to run N virtual-copies sharing memory (quasi N-copy) allowing processor affinity to isolate IRQs and hardware caching.
2. Small memory footprint accomplished by the web-server architecture and/or sharing data when partitioning, but with the trade off of increased contention.
3. Zero-copy data-transmission to eliminate the time/space cost of copying data in/out of the server.
4. Non-blocking I/O, where possible, with the fastest possible event mechanism to know when I/O can be restarted.
5. Web-server open-file cache (versus OS file-data cache) of pre-accessed file-sizes for reply headers, eliminating repeated `stat`ing for existence and size.
6. Controlling lock contention to shared resources when partitioning.

The following sections describe how the μKnot implementation provides these design features.

4.1.1 Web-server Partitioning

Previous multi-core web-server studies show performance benefits by pinning NIC interrupt handlers to different processors and scheduling the web-server process handling requests from these NICs on the same processor [1, 24, 10]. Hence, the process and interrupt affinities are *aligned* improving cache misses, pipeline flushes and locking. Further alignment is possible by having the clients use explicit subnets so threads processing a request only execute on the same CPU handling the network interrupts for the subnet associated with the request. Alignment is accomplished in μC++ by *partitioning* the execution environment into multiple clusters, where each cluster has its own set of UTs and KTs, which execute through a separate ready queue (see Section 2.1.4). This partition isolation ensures threads servicing requests can only execute in an appropriately aligned environment.

4.1.2 Small Footprint

When serving static content, small memory footprint is important. The larger the web server, the less space is available in the OS file-system cache, which is used directly by `sendfile` for zero-copy context-sending. For this reason, a well-written copying web-server is never competitive with a well-written zero-copy web-server.

For a UTPC web-server, the two major implementation components with the greatest affect on memory footprint are accessed thread stack and open-file cache (if present). Reducing memory for these two components involves correctness and a tension between space versus time.

The thread stack is the largest space contributor for any UTPC at high request rate, because there can be tens-of-thousands of simultaneous requests resulting in a corresponding number of simultaneous threads. For example, given a 50K request rate and 16K of accessed thread stack, the resident thread footprint is 819M, which is approximately 1/2 to 1/4 of the total memory for the 2GB and 4GB hardware configurations. Hence, an aggressive attempt was made to reduce the request-thread's stack access.

The minimum thread stack-size is the maximum *dynamic* call-depth, which can be less than the maximum *static* call-depth. However, determining the maximum dynamic call-depth with separately-compiled library-routines (e.g., `sendfile`) is very difficult. Augmenting the calling convention to locate the maximal stack size is also very difficult because local stack allocations are not accounted for. The fallback is an ad-hoc approach that is unsafe in general, but sufficient to gather results and draw general conclusions.

The ad-hoc approach for reducing the stack size starts by compiling μKnot in the μC++ debug-mode to include a read-only guard page at the end of each user-thread stack. Then an experiment is run in GDB to see if there is a segment fault when the stack steps onto the guard page. When there is a failure, the stack trace is printed for the failing dynamic call-chain. All the calls are examined to see if any of the frames can be reduced by removing/reducing local variables. If a reduction is possible, the stack size is reduced slightly and the experiment is run again. This process is repeated until failures are unreducible, meaning the last working dynamic call-chain is likely the maximal one. Using this approach, the stack for a request thread was reduced to 2.5K, while still resulting in a correctly executing experiment. Clearly, this approach is fragile but provides an upper bound on performance for the μKnot UTPC web-server.

Using this technique, the following changes were made to reduce the stack size of requester threads in μKnot.

1. The μC++ non-blocking socket I/O wrappers were switched from exceptions to return codes for errors. Normal μC++ behaviour for a server error (e.g., client drops a connection because it times out) is for the μC++ non-blocking `sendfile` wrapper to raise an exception, which unwinds the stack to a handler in the requester thread. However, creating, propagating and catching an exception significantly increased the maximal dynamic-call depth and involved some large stack-frames. Note, I/O errors are rare events so the performance cost of the exception handling is insignificant, but each requester-thread's stack has to have sufficient space to handle these rare worst-case events.

2. A call to `snprint` for generating the reply request was replaced with bespoke code because `snprint` uses a surprising amount of stack space.

3. Local variables where moved to the UT's object from the UT's stack (`main`) (class versus member variables). Moving the location of this memory versus the amount, gave better control of the stack reduction, i.e., if data does not need to be on the stack, move it elsewhere.

4. A feature to give a thread its stack storage was used versus having the thread allocate its own stack. All the thread stacks were than preallocated in an array, and each array element (fixed-sized stack) was given to a thread. This approach packs the stack space as tightly as possible, with no intervening heap padding or extra heap management fields, which adds up when there are tens of thousands of threads.

5. All signalling for time-slicing and trace analysis (printing running experiment statistics at fixed intervals) is disabled because signal delivery occurs on the current executing thread's stack and UNIX-signal preemption takes significant stack space as all the *process* state is saved on the stack. It is impossible to use separate signal stacks for preemption because the preemptions can nest, destroying the previous preemption state; per-thread signal-stacks are too memory expensive.

6. I/O polling for active events is done is μC++ by nominating a I/O blocking thread (one that received a would-block from an I/O operation) to perform the polling. This thread cycles through polling and yielding, where yielding puts it on the ready queue with all requester threads. Hence, there is convenient delay between polls, until the I/O poller thread gets to the front of the ready queue, to allow the active fileset to build before the next poll. Polling too often is expensive because the active file set returned is very sparse; hence, the ready-queue delay means the poll finds 30-90 events in the active fileset and restarts those requester threads to retry their I/O operation. If the I/O poller's event is satisfied, it nominates another I/O-blocked thread to take over polling.

   The problem is that polling makes a number of system calls and does some complex analyse, both requiring a significant stack-size to ensure safe execution. Since the requester threads have their stacks reduce to a minimum, it is now unsafe for them to poll. To safely poll, the I/O poller thread resumes (context switches) to a coroutine with a large stack, and the coroutine does the polling on its larger stack with the I/O poller's thread and returns when the I/O poller's event is active, at which point another I/O poller has been nominated and it does the same trick with the coroutine. The cost is the two context switches to/from the polling coroutine from the I/O poller task. Note, the I/O-poller coroutine is created once at the start of the μC++ runtime and reused for all I/O polling.

   The key benefit of borrowing a delayed thread for the I/O poller versus a dedicated I/O poller thread is when there is no pending I/O. For a dedicated I/O poller, there is the question of spinning or blocking or both, when there is no work. For a borrowed I/O poller, there are simply no pending threads. Multiple I/O poller are possible, but one was sufficient for μKnot executing the experiments.

7. The default stack-size for administrative tasks is reduced because they are only used for simple operations with the single-purpose web-server.

Note all of these changes are internal to μKnot and μC++, providing additional performance benefits; application programs are unchanged. Some of these stack-growth issues would be handled automatically using split stacks to allow UT stacks to grow dynamically because larger stacks are only needed when transient effects occur, like raising exceptions or delivering a signal.

4.1.3 Zero-copy

μKnot uses `sendfile` for all static requests. Note, `sendfile` is a blocking system call for static content because it takes two FDs, where the input file is the disk file and the output file is the network socket; hence, if the input is not in the file-system cache, `sendfile` blocks the KT. To mitigate these blocking calls, μKnot provides a tunable pool of KTs, allowing up to N simultaneous blocking calls before the web-server blocks.

Experience from tuning other web-servers is that a separate accepting-thread can cause a cascade of problems when all the content I/O blocks, and the accepting thread does not throttle itself. Once the network/disks are saturated, it is better to drop client requests rather than drive the web-server into a failure by over-accepting requests when the server cannot keep up.

4.1.4 Non-blocking I/O

The OS kernel manages all disk and network I/O, and provides system calls to create/open files/-sockets and read/write to/from these I/O devices. In general, all I/O operations are blocking because an application must interact with the OS to establish/destroy an I/O connection and receive/transmit data through it. However, blocking the application misses opportunities for parallelism. Even if an I/O application is sequential on a single processor, it is actually interacting concurrently with multiple parallel components on the computer, e.g., multiple disk and network controllers using DMA. To take advantage of this parallelism, an OS presents the concept of *non-blocking I/O*, so an application can start multiple I/O events, while concurrently processing the results from completed events. This capability is exploited by an event-based web-server. The asynchrony between the start of the I/O operations and its completion provides the concurrency. The difficult problem is testing for completed events and matching these events with the corresponding actions in the web server.

To manage I/O, the OS retains internal state about active connections and the data being transmitted to/from the connection for an application. Different OSs provide different mechanisms for non-blocking I/O, and may even provide multiple mechanisms, making it difficult for a programmer to choose among them. In many cases, an application has to shadow the I/O activity to know what asynchronous operations are outstanding and which ones have completed. The amount of shadow information retained by the application, plus the number of kernel-boundary crossings to check for completed I/O varies significantly with the OS mechanism. Minimizing the duplication, transmission, and checking of shadow information has the potential to increase performance.

The three main system calls for registering non-blocking FDs and handling pending I/O requests are: `select`, `poll`, and `epoll` (`kqueue` in BSD). The system-calls `select` and `poll` work by taking a set of open (usually socket) FDs, and returning information about which FDs are ready for read, writing, or have triggered exceptions. Both `select`/`poll` pass FD interest-sets that are proportional to the number of active FDs for dense file sets, e.g., if there are 30K active client requests, there are 30K active FDs, and so up to 30K units of FD set cross the kernel boundary bidirectionally. Even for for sparse FD interest-sets, there are limited capabilities to reduce the amount of copying. Finally, for each call to `select` and `poll`, the kernel inspects the entire set of file descriptors, even when the set is sparse. Banga et al. [3] compare the performance of `select` and `poll` and conclude that they do not scale very well. However, Gammo et al. [27] claim similar performance for `select`, `poll`, and `epoll` for dense FD sets up to 30K requests per second; only when the FD set is extremely sparse is there a 79% performance reduction for `select` and `poll`.

The system-call `epoll` was proposed by Banga et al. [3] as a more scalable polling mechanisms for FD interest-sets. `epoll` is often *purported* to be the simplest and cheapest mechanism to perform non-blocking I/O operations in Linux. The reason `epoll` is more performant is the reduction in the amount of information that crosses the kernel boundary when the application polls for completed I/O events. The approach is to remove shadow information from the application, so the FD interest-set (or equivalent) only resides in the kernel. With `epoll`, an FD is registered once, which puts it into the kernel interest-set. Hence, there is a system call after open and before close to register/deregister the FD. The user application then calls an event-polling routine to fetch any events associated with registered FDs that have become active, i.e., these FDs might not block when the I/O operation is performed again (no guarantees in a concurrent system). Hence, there is only a small set of active FDs moving unidirectionally across the kernel boundary. The advantage is scaling as the cost of event-polling dependents upon the number of active FDs rather than the total number of FDs. Furthermore, it is possible for `epoll` to implicitly re-enable an FD after an I/O operation fails with a would-block result (edge trigger), rather than requiring an explicit `epoll` call to reactivate interest in the FD.

However, `epoll` has problems. It is prone to spurious activations, where an FD is returned from a poll that is not ready. Also, `epoll` does not work with regular files, shell redirection to a regular file, and some other cases. In these situations, the FD does not register and blocking occurs on I/O, which should not occur in these non-blocking I/O cases. This latter issue is a major problem in porting `epoll` into μC++, because μC++ is a full-service concurrency-package, requiring all previously non-blocking I/O operations to remain non-blocking. Since `epoll` cannot handle all cases, it meant using two techniques to handle non-blocking I/O, such as `select` and `epoll`. However, this design presents problems, where the most difficult problem is handling the point where μC++ has no work and needs to block until an FD becomes active. However, `select` and `epoll` have separate blocking mechanisms and are unaware of each others FD sets. After some effort, I abandoned porting `epoll` into μC++ because to do it properly is beyond the scope of the thesis. Note, other web server using `epoll` do not have to provide full-service non-blocking I/O, which is why it is possible for them to use it.

Therefore, all μKnot experiments are run using the uniform `select` system-call, which works for all non-blocking I/O on UNIX. In comparison to other web-servers using `epoll`, there is little performance difference; where there is a difference, it is difficult to tell if it results from memory

or `select`. This observation matches with the result from Gammo et al. [27], that `select` is performant with `epoll` for dense FD sets, even though there is significant information crossing the kernel boundary

4.1.5 File-name Cache and Contention

A number of different hash-table designs and locking techniques were examined (see Appendix A), but there was only marginal performance benefits (if at all). I observed that regardless of the hash-table design, μKnot, with an in-memory Zipf workload, where clients access the majority of the 21,396 file-set during an experiment[1], the number of cache misses drops to 0 (100% hit-rate) within 5-15 seconds (experiments ran 300 seconds or longer). The duration to zero cache-misses depends on the order files are presented by the clients versus the quality of the hash table. The recommendation is that super optimizing the open-file cache is unnecessary because most high-performance web-servers only use about 5%-10% of the processor cycles, where only a small percentage of this time is spent in the open-file cache; output from `vmstat` shows `sendfile` and servicing IRQs occupies the bulk of processor time in an experiment. The conclusion is that any *well-written* hash-table should provide adequate size/speed trade-offs for the open-file cache, where the emphasis should be reduced size over faster speed with low overall read-contention.

Contention on the hash table by multiple threads performing lookup can be a problem. Again, a few locking and data reorganization techniques were examined. Like the hash table itself, the locking approaches did not make any significant difference. Even at high request rates of 70K requests per second, the number of active requests in the server is usually less than 10K, meaning the contention on the hash table is fairly low. As above, the conclusion is that any *well-written* readers-writer locking mechanism should provide adequate response for concurrent access to the open-file cache, where the emphasis should be optimizing readers over writers.

4.2 New Kernel

The experimental setup and hardware for testing in Chapter 3 is virtually the same as that used by Harji [33], except Harji used Linux kernel 2.6.24-3 SMP in 32-bit mode. This old kernel had excellent performance with repeatable results. My work uses the newer Linux kernel 3.2.0-126 SMP kernel in 64-bit mode. The newer kernel generated results that matched well with the overlapping web-servers in Harji's prior work for in-memory workloads. However, the newer kernel results were significantly slower for disk I/O. In fact, it was necessary to increase the amount of memory from Harji's 2GB to 2.4GB to get reasonable results. Even after many weeks of tuning web-servers, it was impossible to achieve throughput approaching Harji's, results in a reduction of 50%. Furthermore, during tuning, some of the web servers behaved strangely, with large jitter, non-repeatable results, and some failures. It was beyond the scope of this thesis to

[1] The experimental file-set is based on the requests processed by the server. Because a client request can timeout at high request rates, these failed requests, along with any subsequent requests in a persistent session, are not sent by the server. Hence, not all files in a client trace are processed by the server, meaning the effective file-set experienced by the server varies non-deterministically from run to run and with the request rate.

determine if these problems are issues with the web server and/or the OS. Having said that, it is the case that multiple web servers experienced problems so that significantly points to a cross web-server OS issue.

Attempts to switch to a current kernel via upgrading Ubuntu resulted in an unexplainable performance ceiling at approximately 5 Gbps for all web-servers that previously exceeded this throughput. An attempt was made to harmonize the old Harji kernel controls under `/proc` (where applicable) with the new kernels, but no amount of updating values produced any benefit. I quickly gave up trying to understand the performance regression and pressed on with the updated OS-kernel. I discovered that using the latest Linux kernel for running experiments is not always the best approach [34].

Chapter 5

Conclusion

5.1 Summary

With the processor scaling-limits being reached, there is a higher emphasis on writing high-performance, parallel code that can take advantage of multiple cores, while being easy to maintain. The primary aim of this thesis is to study the performance characteristics of concurrent programming-models in the context of high-performance web-servers. This work examined several approaches used to handle concurrency in the face of blocking I/O operations – event-driven, asynchronous, 1:1 kernel-level multi-threading, and M:N user-level multi-threading. The goal of this thesis is to provide a comprehensive view of these approaches in both a memory-constrained and high-memory environment. In addition, a highly-concurrent UTPC web-server, μKnot, is constructed from scratch in the μC++ runtime environment to study and evaluate the performance of user-level threading.

To compare the diverse web-server architectures, a simple experiment is used to fetch static context from the web server to clients at various request rates. An important lesson learnt is that web-server tuning is crucial to achieve good performance for the experiments conducted in this thesis. Significant effort was taken to ensure the comparison of the aforementioned approaches to concurrency happened on an even footing, by carefully choosing representative web-servers and tuning them appropriately, while keeping "bells and whistles" in the servers to a bare minimum. Unfortunately, in the under-provisioned environment (2GB), the OS played a significant negative factor, distorting the results so it is hard to draw strong conclusions in that context.

The results in the thesis demonstrate that some UTPC web-servers (Apache, goserver, gofasthhtp, YAWS) are uncompetitive with event and pipeline web-servers, reenforcing the stereotype against TPC web-servers. However, academic μKnot web-server is competitive with the two academic web-servers μServer and Watpipe and the industrial-grade NGINX web-server in both the 2GB and 4GB domains. Furthermore, the μKnot/μC++ server and user-level threading provide a natural abstraction, which makes it easy to write the concurrent web server in a natural way. The μKnot web-server accomplished this feat by following the list of design principles developed from Harji's work. In addition, experiments show that asynchronous (event-driven) approaches to concurrency are highly memory efficient, but the trade-off occurs in terms of

higher programmer burden and large, complex, state-machines. The objective of this thesis is not to offer a conclusion on which type of approach is better, it is to demonstrate that all of the above paradigms can be achieve competitive performance, so none can be ruled out. By outlining the trade-offs among the approaches, this work hopes to inform better decisions when choosing a runtime paradigm for a concurrent application.

5.2 Future Work

Blocking thread-pool: An interesting iteration of μKnot is to use two KT clusters: one for the server and the other to handle blocking I/O operations. The blocking cluster has a pool of N KTs that can block independently from the KTs executing the server (as for event-based servers). For example, before a user thread calls `sendfile`, it migrates to the blocking cluster to make the call, and after the call completes, it migrates back to the server cluster.

Latency Measurements: Throughput is used as the primary means to quantify servers in this work. Latency measurements were not used for a few reasons. In a typical distributed system (of which servers are usually a part), a request is typically dominated by wide-area-network latencies, rather than the server latency. Additionally, this work is not concerned with raw throughput and latency, but rather the throughput of each server architecture relative to the others being compared. In order to ensure that latencies did not factor into the work, a 10 second timeout was chosen for requests, which caps latency measurements. While latency is an important characteristic of a web-server, it is non-trivial to compare the servers on two dimensions. A possible extension of this work could be to pick a reasonable latency target and filter requests that do not meet the target. This kind of workload would also model real-world scenarios where requests are bound by service-level-agreements (SLAs) on latencies.

Bypassing the kernel: One interesting follow-up of this work is to evaluate the performance of user-level concurrency in combination with other approaches to reduce or eliminate kernel involvement in a distributed system. For instance, RDMA [46, 40] has emerged as a paradigm to bypass the kernel and access memory in remote hosts using the NIC. Additionally, as disk read/write latencies approach the latency of traps and kernel context switches, efforts to directly write to the disk from user-space have emerged [73, 14]. Many distributed systems offered as cloud services optimize for large amounts of disk and network I/O. The work in this thesis can be used towards designing a user-level stack of applications, or even a highly specialized operating system that performs parallel I/O at high efficiency using one or more of the techniques described above.

References

[1] Vaijayanthimala Anand and Bill Hartner. TCPIP network stack performance in Linux kernel 2.4 and 2.5. In *Proc. of the 4th Ottawa Linux Symp.*, June 2002.

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

[3] Gaurav Banga, Jeffrey C Mogul, Peter Druschel, et al. A scalable and explicit event delivery mechanism for unix. In *USENIX Annual Technical Conference, General Track*, pages 253–265, 1999.

[4] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 151–160. ACM, 1998.

[5] Saman Barghi. *Improving the Performance of User-level Runtime Systems for Concurrent Applications*. PhD thesis, School of Computer Science, University of Waterloo, September 2018. https://uwspace.uwaterloo.ca/handle/10012/13935.

[6] Jeff Barr. AWS auto scaling, January 2018. https://aws.amazon.com/autoscaling.

[7] David Abrahams Beman Dawes and Rene Rivera. The Boost C++ Library. https://www.boost.org, 2017.

[8] Stephen J. Bigelow. Don't waste valuable data center resources on overprovisioning, May 2016. https://searchservervirtualization.techtarget.com/tip/Dont-waste-valuable-data-center-resources-on-overprovisioning.

[9] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[10] Tim Brecht, G. (John) Janakiraman, Brian Lynn, Vikram Saletore, and Yoshio Turner. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. In *EuroSys'06*, pages 265–278, Leuven, Belgium, April 2006.

[11] Walter Bright and Andrei Alexandrescu. *D Programming Language*. Digital Mars, 2016. http://dlang.org/spec/spec.html.

[12] Peter A. Buhr. *Understanding Control Flow: Concurrent Programming using μC++*. Springer, Switzerland, 2016.

[13] Peter A. Buhr. *μC++ Annotated Reference Manual, Version 7.0.0*. University of Waterloo, September 2018. https://plg.uwaterloo.ca/~usystem/pub/uSystem/uC++.pdf.

[14] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[15] Constantinos Christofi, George Michael, Pedro Trancoso, and Paraskevas Evripidou. Exploring HPC parallelism with data-driven multithreating. In *2012 Data-Flow Execution Models for Extreme Scale Computing*, pages 10–17, September 2012.

[16] Intel Corporation. Intel hyper-threading technology. https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, 2018.

[17] Standard Performance Evaluation Corporation. Specweb99 benchmark. http://www.specbench.org/osg/web99, 1999.

[18] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

[19] Beman Dawes, David Abrahams, and Rene Rivera. The boost c++ library: Intrusive and non-intrusive containers. https://www.boost.org/doc/libs/1_43_0/doc/html/intrusive/intrusive_vs_nontrusive.html, 2011.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[21] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19, September 1997.

[22] Erlang AB. *Erlang/OTP System Documentation 8.1*, September 2016. http://erlang.org/doc/pdf/otp-system-documentation.pdf.

[23] Facebook. Folly: Facebook open-source library, 2017.

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

[25] The Apache Software Foundation. Apache : HTTP Server Project. http://httpd.apache.org, 2018.

[26] The Apache Software Foundation. Apache : HTTP Server Project, VERSION 2.4 - Directive Quick Reference To Modules. http://httpd.apache.org/docs/2.4/mod/quickreference.html, 2018.

[27] 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.

[28] Golang. *Package http*. Google, 2019. https://golang.org/pkg/net/http.

[29] Google web server. https://en.wikipedia.org/wiki/Google_Web_Server.

[30] Robert Graham. The c10m problem. http://c10m.robertgraham.com, 2017.

[31] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, 2009. http://golang.org/ref/spec.

[32] Mark Gritter. Data dive: Vm sizes in the real world, May 2016. https://www.tintri.com/blog/2016/05/data-dive-vm-sizes-real-world.

[33] Ashif Harji. *Performance Comparison of Uniprocessor and Multiprocessor Web Server Architectures*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, February 2010. https://uwspace.uwaterloo.ca/handle/10012/5040.

[34] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. Our troubles with Linux kernel upgrades and why you should care. *SIGOPS Oper. Syst. Rev.*, 47(2):66–72, July 2013.

[35] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[36] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[37] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Not.*, 27(5):T1–53, May 1992.

[38] husobee. http implementation fasthttp in golang. https://husobee.github.io/golang/fasthttp/2016/06/23/golang-fasthttp.html, 2016.

[39] Internet Live Stats. Google statistics. http://www.internetlivestats.com, 2018.

[40] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[41] Dan Kegel. The c10k problem. http://www.kegel.com/c10k.html, 2017.

[42] Ari Luotonen, Henrik Frystyk Nielsen, and Tim Berners-Lee. Cern/w3c httpd web-server codebase repository. https://www.w3.org/Daemon, 1996.

[43] Nick Martin. Overprovisioning vms may be safe, but it isn't sound, May 2014. https://searchservervirtualization.techtarget.com/feature/Overprovisioning-VMs-may-be-safe-but-it-isnt-sound.

[44] John McCutchan. Gamasutra : The Art and Business of Making Games. https://www.gamasutra.com/view/news/128568/InDepth_Intrusive_Lists.php, 2011.

[45] Microsoft. Fibres – About Processes & Threads. https://docs.microsoft.com/en-us/windows/desktop/procthread/fibers, 2018.

[46] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided {RDMA} reads to ld a fast, cpu-efficient key-value store. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 103–114, 2013.

[47] David Mosberger, Martin Arlitt, Ted Bullock, Tai Jin, Stephane Eranian, Richard Carter, Andrew Hately, and Adrian Chadd. Http load generator. https://github.com/httperf/httperf, November 2018.

[48] NetCraft. June 2018 Web Server Survey. https://news.netcraft.com/archives/2018/06/13/june-2018-web-server-survey.html, 2018.

[49] NGINX. Challenged by the growth of the world wide web: Nginx vs. apache: Our view of a decade-old question. https://www.nginx.com/blog/nginx-vs-apache-our-view/, 2014.

[50] NGINX. Inside NGINX: How We Designed for Performance & Scale. https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale, 2015.

[51] NGINX. Tuning Your NGINX configuration. https://www.nginx.com/blog/tuning-nginx/, 2015.

[52] NGINX. Serving static content. https://docs.nginx.com/nginx/admin-guide/web-server/serving-static-content/, 2019.

[53] & Niels Provos Nick Mathewson, Azat Khuzhin. Libevent – an event notification library. http://libevent.org/, 2017.

[54] Nutanix. Understanding web-scale properties. https://www.nutanix.com/2014/03/11/understanding-web-scale-properties/, 2014.

[55] John Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.

[56] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. of the USENIX Annual Tech. Conf.*, 1999.

[57] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference, General Track*, pages 199–212, 1999.

[58] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla. Comparing the performance of web server architectures. *SIGOPS Oper. Syst. Rev.*, 41(3):231–243, March 2007.

[59] Susanna Pelagatti. *Structured development of parallel programs*, volume 102. Taylor & Francis Abington, 1998.

[60] Marc Pérache, Hervé Jourdren, and Raymond Namyst. Mpc: A unified parallel runtime for clusters of NUMA machines. In *Euro-Par 2008*, volume 5168 of *Lecture Notes in Computer Science*, pages 329–342. Springer, Berlin, Heidelberg, 2008.

[61] Quasar. *Quasar Documentation, Release 0.8.0*. Parallel Universe, http://docs.paralleluniverse.co/quasar, 2018.

[62] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[63] David B Skillicorn. *Foundations of parallel programming*. Cambridge University Press, 2005.

[64] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Unix network programming volume 1: The sockets networking api, 2003.

[65] Xianda Sun. Concurrent high-performance persistent hash table in Java. Master's thesis, School of Computer Sc., University of Waterloo, 2015. https://uwspace.uwaterloo.ca/handle/10012/10013.

[66] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. http://www.gotw.ca/publications/concurrency-ddj.htm, 2005. originally Dr. Dobb's Journal 30(3).

[67] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[68] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[69] Aliaksandr Valialkin. Go fasthttp. https://github.com/valyala/fasthttp.

[70] Aliaksandr Valialkin. Gofasthttp server benchmarks. https://github.com/valyala/fasthttp/blob/master/README.md#http-server-performance-comparison-with-nethttp, 2018.

[71] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, pages 19–24, 2003.

[72] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. *SIGOPS Oper. Syst. Rev.*, 37(5):268–281, October 2003.

[73] Benjamin Walker. Spdk: Building blocks for scalable, high performance storage applications. In *Storage Developer Conference. SNIA*, 2016.

[74] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, October 2001.

[75] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *International Symposium on Parallel and Distributed Processing*, Miami, FL, USA, April 2008. IEEE.

[76] WikiChip. AMD simultaneous multithreading. https://en.wikichip.org/wiki/amd/microarchitectures/zen#Simultaneous_MultiThreading_.28SMT.29, January 2019.

[77] Anthony Williams and Vicente J. Botet Escriba. Boost thread library. https://www.boost.org/doc/libs/1_61_0/doc/html/thread.html, 2015.

[78] Simon Wilson. Node.js is genuinely exciting. https://simonwillison.net/2009/Nov/23/node, 2009.

[79] Patrick Wyatt. Avoiding Game Crashes Related to Linked Lists. https://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists, 2012.

[80] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proc. of the USENIX Annual Tech. Conf.*, pages 239–252, June 2003.

Appendix A

HTTP Response Cache

Workload plays an important role in determining caching policy. For instance, LRU (Least-Recently-Used) caches are more useful when requests to a single item are temporally adjacent. This benefit is because LRU caches evict entries that have not been accessed recently. (Note, the definition of "recently" varies, and typically is a set based on heuristics of workloads). A similar type of cache-eviction policy is Least-Frequently-Used (LFU) cache. As the name suggests, this policy optimizes cache performance for workloads containing reads that are skewed towards a smaller subset of keys, and are not necessarily closely temporally spaced. Web-server traffic typically follows a power-law distribution, as mentioned previously, which means there is a subset of "popular" items over bursts of time, causing accesses with high temporal adjacency. Therefore, an LRU cache is most appropriate to implement the HTTP-metadata cache in μKnot.

A.1 Mutual Exclusion

While it is possible to implement a cache with several different data structures, the obvious choose is a hash table because of the average O(1) lookup. Its worst case, O(N), can be made small via a combination of a good hash function for character strings (file names) and over-provisioning the number of hash buckets to reduce collisions. Following these two simple rules resulted in an average hash-chain length of > 1.5 for the experimental fileset, where the number of buckets is 20K and each bucket is small (pointer to hash chain, fast check, pointer to file name).

Choosing reasonable locking for concurrent access of the shared cache is important to performance. For global bucket locking, a readers-writer (RW) lock is best to take advantage of workloads with few evictions (in-memory), i.e., most operations are reads. For per bucket locking, a simple spin lock suffices [65] because contention is virtually zero when collisions are low, which is true for the hash table above. While hand-over-hand locking implementations provide even higher concurrency of operations, such as walking hash chains, this additional concurrent is unnecessary for short hash-chains. An alternative approach for protecting each has bucket is to use a lock-free queue. However, previous work [65] has shown a simple spinlock per bucket performs better than a lock-free (Java) queue.

Tangential to the chosen locking mechanism, concurrency of read operations to the cache can be handled in one of two ways – by copy, or by reference. In the copy-approach, a key is read atomically by copying out the value out of the hash table. This approach implies the reader of the cache may have a stale value at some future point caused by a later write to that key. Writes are strongly consistent, meaning that they are done by reference. In the reference-approach, a pointer to the key is acquired and held until the read or write operation has completed. This approach ensures consistency of reads, but moves less data between the cache and its clients. Since the μKnot cache cares less about consistency and more about optimizing performance, and given that the key-value pairs are small (a few tens of bytes), the copy-approach is chosen.

A.2 Intrusive List

One of the most common data structures for a cache is a **L**east **R**ecently **U**sed ordered list. Therefore, an optimized LRU list implementation can significantly reduce latency of cache accesses, while improving throughput. This section explores the benefits of using "intrusive" lists [19, 44, 79] over standard linked-list design in implementation of an LRU cache.

Figure A.2 shows the difference between the two list implementations in pseudocode. In the standard list implementation, data and list are uncoupled, so the data is independent from the list. In an intrusive list, the link nodes are embedded into the data object, hence the name *intrusive*. Figure A.1 shows the memory layout of the two types of lists. Intrusive lists are beneficial for the following reasons:

Memory : Since links are embedded into the data, it is unnecessary to allocate/deallocate storage for a data copy in the list node. In cases, where the data copies are always consistent (i.e., the original and copy are not mutated independently), having multiple copies only increases the memory footprint. Additionally, non-intrusive lists require copy or move constructors to move the data into the list node, which means extra runtime cost to move the data, and non-copyable or immutable objects cannot be stored in non-intrusive lists. Therefore, intrusive lists are a good choice for use in a high-speed meta-data cache to optimize for speed and memory.

Cache Thrashing : As can be seen from the memory layout diagram in Figure A.1, it takes only a single pointer indirection to get to an object in an intrusive list, compared to the two pointer hops to access the object in the non-intrusive case. The additional pointer reference *per access*, for millions of possible accesses (in the case of the μKnot metadata cache) can lead to cache thrashing, which can cause pipeline stalls and performance bottlenecks. Furthermore, when the data and list node(s) in an intrusive list can share a cacheline, it makes the structure manipulation faster and more cache-friendly.

Expressibility : Unlike non-intrusive lists, data objects can participate in multiple intrusive lists at a time. A single object can hold multiple intrusive list nodes (while retaining cache-friendliness), making it especially attractive as a means to express applications like LRU caches, which utilize multiple lists. For instance, in the current implementation, a single hash bucket entry holds two list nodes – one for the bucket's collision chain, and the other for the higher-level LRU list. Though the cache and underlying hashmap are not cleanly separated in such a design,

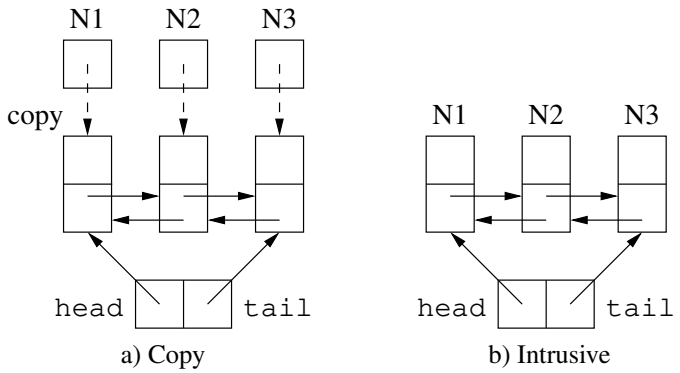


Figure A.1: Memory Layout for copy and intrusive lists

```
struct Node {
    Type data;
};
struct ListNode {
    Type data;
    Node *next, *prev;
};
struct List {
    ListNode *head, *tail;
};
```

a) Copy

```
struct Node {
    Type data;
    Node *next, *prev;
};


struct IList {
    Node *head, *tail;
}
```

b) Intrusive

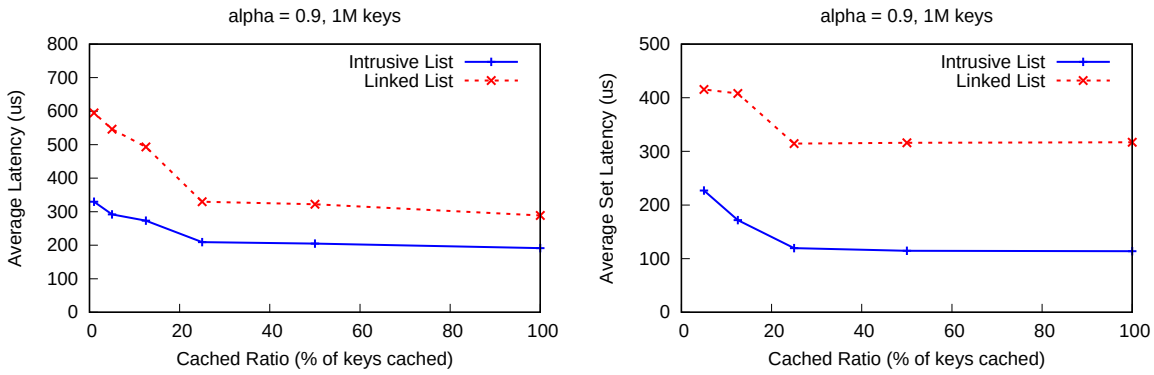
Figure A.2: Pseudocode for copy and intrustive lists

the tight-knit nature the two lists leads to lower cache and memory overheads, when compared to non-intrusive lists. Therefore, in the metadata cache implementation uses intrusive lists.

Reliability & Complexity : When deleting an object, its destructor can verify it is not currently on any list(s) and possibly remove it from these, if appropriate, leading to fewer memory bugs and crashes. Intrusive lists also help avoid out-of-memory exceptions when linking items together, since linking and memory allocation are decoupled. Therefore, intrusive lists offer better exception guarantees than their non-intrusive counterparts.

A.3 Evaluation

In order to evaluate the benefits offered by intrusive lists, two atomic LRU caches are built – one utilizing a lightweight intrusive list implementation, and the other using the C++ Standard Library (STD) list. Both implementations use the same underlying hashmap (an STD Unordered Map) to store items, and Read-Write Locks to achieve atomic concurrent access to the underlying data structures. The caches are evaluated with a microbenchmark that mimics HTTP request patterns using a Zipf distribution of accesses. The evaluation is conducted on a machine with 8 Intel



a) End-to-end latency

b) Set latency

Figure A.3: Request latency as cache size is increased. Latency decreases in both implementations, but overall latency is lower in intrusive-list based cache.

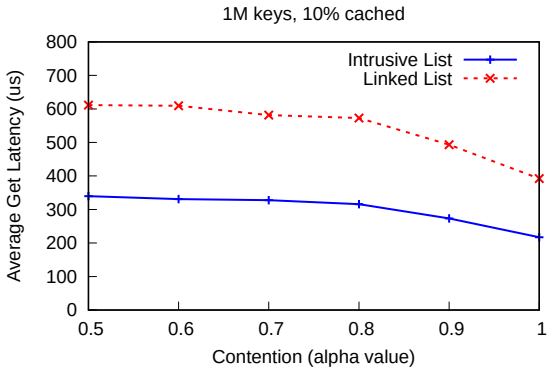


Figure A.4: End-to-end latency as contention ratio is varied.

i7-6700 CPUs, clocked at 3.40GHz each, with 32GB of available RAM, running Ubuntu 16.04 with Linux Kernel v4.13.0-39. Each client thread generates requests from a Zipf distribution of 1 Million keys. Caches testing is performed by varying the ratio of keys cached, the alpha value of the distribution (which affects thread contention on popular keys) and the number of threads issuing requests in parallel.

Figure A.3 a) shows the latency of a Get request as the percentage of cached keys is varied from 1% to 100%, while keeping the contention ratio (alpha value) and request rate constant. Each Get request on the cache takes the following path – if the key is present (cache hit), it is returned immediately, and bumped up to the top of the LRU list. If a cache miss occurs, the key is loaded into the cache and subsequently returned. If the number of keys present in the cache exceeds the allotted number, the LRU used key is evicted and the new key is loaded into the cache. The intrusive list performs markedly better (lower is better) compared to the STD list, due to its cache-friendliness. The importance of a good list implementation in the cache is evident here, since each operation touches the list to move the LRU touched item to the front of the list. Figure A.3 b) shows the latency of the Set and Evict operations, when cache misses and evictions occur.

In order to see the effect of varying contention on cache performance, the end-to-end latency

of Get requests is measured, increasing the alpha parameter of the Zipf distribution while keeping the cache size and request rate constant. Figure A.4 shows the contention has a positive effect on the cache latency, since more items are "popular", leading to a higher hit-rate on the cache (87% for an alpha value of 1.0). However, the overall latency of requests in the intrusive case is lower than the non-intrusive case, showing that the benefits of intrusive lists are retained even when the hit-rate is low (52% for an alpha value of 0.5).

A.4 Summary

Experimental evaluations of different locking techniques showed that complex locking protocols did not provide adequate benefit to justify code complexity. Intrusive lists offered a good optimization to use in the hash map compared to standard list implementations. These observations were integrated into the metadata cache of μKnot by using per-bucket spinlocks and intrusive lists to improve its performance.