Technical Report

UCAM-CL-TR-781
ISSN 1476-2986

Number 781

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Stage scheduling for CPU-intensive servers

Minor E. Gordon

June 2010

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2009 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Jesus College.

ISSN 1476-2986

Stage scheduling for CPU-intensive servers

Minor E. Gordon

Summary

The increasing prevalence of multicore, multiprocessor commodity hardware calls for server software architectures that are cycle-efficient on individual cores and can maximize concurrency across an entire machine. In order to achieve both ends this dissertation advocates stage architectures that put software concurrency foremost and aggressive CPU scheduling that exploits the common structure and runtime behavior of CPU-intensive servers. For these servers user-level scheduling policies that multiplex one kernel thread per physical core can outperform those that utilize pools of worker threads per stage on CPU-intensive workloads. Boosting the hardware efficiency of servers in userspace means a single machine can handle more users without tuning, operating system modifications, or better hardware.

Contents

Chapter 1

Introduction

Structuring server request processing logic as a series of concurrently-executable stages and aggressively scheduling stage execution in userspace with a single kernel thread per core can increase the throughput of a CPU-intensive server running on a multiprocessor machine in comparison to thread-per-connection concurrency and thread pool-per-stage scheduling. The resulting gains in throughput do not come at the expense of significantly increased latency, which is kept under acceptable upper bounds for a given application.

That is my thesis. The remainder of this chapter defines the key terms in the thesis, sets the scope of the dissertation, and outlines its contents.

1.1 Servers

A server is a program that manages access to a shared resource, such as a database, a file system, or a web site. Servers are *reactive*: they receive requests from clients, process each request, and [optionally] produce responses or other results. From a CPU scheduling perspective the most interesting servers are those that

1. **are highly concurrent**, where $n =$ the number of requests being processed concurrently is much greater than $p =$ cores in a multiprocessor machine, with n on the order of thousands or tens of thousands and $2 \leq p \leq 32$ in current hardware

2. **process requests independently** with little or no unavoidable read/write contention between requests

3. **apply a sequence of operations** in request processing that require minimal synchronization between operations

4. **provide "best effort" service** with no hard or soft real-time guarantees

5. **are CPU-bound** or at least CPU-intensive, so greater CPU efficiency will increase the throughput of the server

This class of servers includes web applications that process images and videos uploaded by users; daemons that apply CPU- and memory-intensive algorithms such as statistical inference mail filters [PRH07] or subscription matching [FJL+01] to incoming requests; caching DNS [Moc87] servers [JV06], HTTP [FGM+99] file servers [PBH+07], and others that work entirely from memory; servers that do extensive parsing and serialization of text formats [KS03]; and database servers under read-intensive workloads [HPJ+07].

1.2 Stages

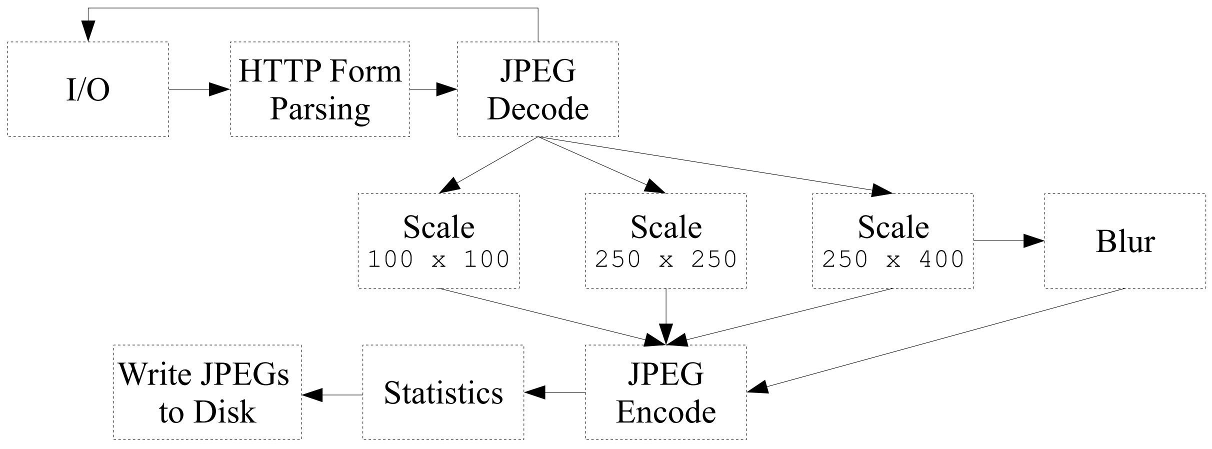
The request processing operations of a server can be divided into *stages*. A stage is an operation that can run simultaneously with another, different operation on a different physical core with minimal or no synchronization between the two operations. Stages have task parallelism [Amd67]. "Minimal synchronization" means that a stage often requires synchronization in order to have work to do, but runs to completion without synchronization in doing the actual work. The initial synchronization often takes the form of signaled queues: a stage dequeues work from a queue, possibly waiting on the queue if it is empty, and a stage generates work for other stages by enqueueing the work into other stages' queues. The interaction patterns between stages can be described by a directed graph, called a *stage graph*.

When server logic is explicitly structured as a series of stages, a CPU scheduler that is aware of these divisions can exploit optimizations that would be unavailable to a scheduler that had less information about the instruction stream under its control.

1.3 An image processing server

Figure 1.1 illustrates a staged server with each stage shown as a separate box. The server is designed to process images uploaded by users via HTTP for subsequent display on a web site. The stage graph for this server consists of an I/O stage for accepting connections, reading HTTP requests from connections, parsing the requests, and writing HTTP responses to connections; a stage for parsing JPEG [Wal91] image data from form-encoded HTTP request bodies; a set of image transforms, such as scaling and blurring, with one transform per stage; and auxiliary stages for collecting statistics and writing output images to disk.

Figure 1.1: Image processing server



The image processing server's design was derived from a Python[1] script that batch processes images uploaded by users of a community site in Germany, `wie-ich.de`. On the real site an Apache [LL02] web server with a Tomcat [BD03] application server back end writes files to a directory on disk, where they are picked up asynchronously by the script. After determining the resolution, metadata, and other parameters of the uploaded image the script spawns separate processes to transform the image using a command line interface to the popular ImageMagick tool[2]. ImageMagick expects files as input, so intermediate results are also written to the file system. The overhead of continually forking processes, decoding and encoding JPEGs many times, and writing all intermediate results to the file system makes this script-based implementation extremely inefficient: it is only able to process 1-2 images per second on a machine with one core. For this reason and because of the tools involved the script works entirely offline.

Processing images online while the user is waiting for an HTTP response would require much higher throughput and much lower latency from the servers and the script. This was the impetus behind the staged server implementation, which subsumes the functions of Apache, Tomcat, and the Python script. In the staged server all of the image transforms as well as the auxiliary functions (accepting uploads, writing the final outputs to disk) are executed in the web server. Only the final output images are written to the file system. Eliminating most of the overhead in the application dramatically increases its performance and makes the server a prototypical example of the class of CPU-bound servers described above.

Given a staged server such as this, my goal is to maximize its throughput, where throughput is defined as the number of requests a server can process to completion in a given period of time[3] while maintaining acceptable response times[4]. The number of requests handled by a server is directly proportional to the number of clients, with anywhere from one to a few dozen requests per client. The number of clients is in turn proportional to the number of users, often in a 1:1 ratio. These relationships imply that a server with higher throughput can support more users. When a server can support more users without upgrading hardware, fewer or cheaper machines are required to support the same total number of users. Less hardware translates to lower setup and maintenance costs.

1.4 Optimizing CPU scheduling for throughput

Throughput-oriented CPU scheduling is crucial to the performance of many compute-intensive applications, including those in scientific computing [YRP+07] and information retrieval [YRP+07]. Applications in these domains have much in common with CPU-intensive servers: their interactions with human users are usually limited to initial inputs (requests, data); they tend to have relatively few code paths compared to end user-facing programs; they have inherent data parallelism [HGLS86] and task parallelism [Amd67],

[1]`http://www.python.org/`

[2]`http://www.imagemagick.org/`

[3]The period for measuring throughput is on the order of seconds or minutes rather than the hours or days of e.g. [ABC+05].

[4]The definition of "acceptable response times" depends on the application and many other variables, but see `http://www.useit.com/papers/responsetime.html` or [Nie94] for a brief overview of response times from a user's perspective.

the latter often taking the form of pipelines; and they rely heavily on the branch predictors, data and instruction caches, and other features of modern processors to maximize throughput on individual cores. A well-designed CPU scheduler for servers can exploit all of these characteristics through targeted optimizations that improve CPU efficiency on individual cores by batching and prioritizing work as well as optimizations that increase parallelism across cores.

Batching

A CPU scheduler can raise the throughput of a server by exploiting batching effects. By applying the same operation with the same code on the same processor to different requests one after the other, the scheduler can increase the instruction cache hit rate, which decreases wasted (stalled) CPU time and thus improves the hardware efficiency of the server. If an operation repeatedly accesses a data structure in memory batching the operation can also increase the data cache hit rate. Aggressively exploiting L1 and L2 caches through spatial and temporal locality of memory references is essential to getting good performance out of modern processors, where loading a word from primary storage can take hundreds of cycles. Limiting the length of operation code paths and the size of data structures to fit in cache (as in loop tiling [IT88]) is also important.

Prioritization

A CPU scheduler can increase a server's total throughput by ensuring that some request processing operations have priority over others. Accepting new connections to a server might have priority over reading existing connections, because accepting more connections under heavy load can prevent clients from going into TCP backoff or users initiating retries [BPG04]. A scheduler can also limit the CPU time of low-priority operations without limiting other, higher-priority operations on the same request. For example, an auxiliary operation that generated statistics from input data might be given very low priority compared to actually receiving the input and sending a response.

When it comes to prioritizing one request over another, the Shortest Remaining Processing Time (SRPT) queue discipline is known to be optimal in terms of both throughput and latency for most service and arrival rate distributions [HBBSA01]. The downside of SRPT is that is a clairvoyant discipline: the scheduler must know the remaining processing times of jobs (requests) in order to prefer jobs that are closer to completion than other jobs. For most online applications, including servers, predicting the remaining processing time of a job is difficult or impossible to do with absolute certainty. However, a server can approximate this heuristic by executing stages at the end of a pipeline-like stage graph in preference to those at the beginning, on the assumption that the relative position of an operation is indicative of the amount of processing time required to complete all of the server's operations. The imposition of explicit signaled queues between stages also opens up a number of possibilities for size- or age-based work reordering.

Parallelism

The stages of a server are usually defined by function and can vary in granularity: image decoding, cache lookups, or XML parsing. Knowing what operations can execute concurrently simplifies the job of a multiprocessor scheduler, because it can guarantee that no operation will block or spin waiting for the results of another operation once the first operation has started. Having fixed code paths that perform only one or a few variations of an operation also makes the runtime cost of operations easier to predict, which enables more accurate resource control.

When the stage graph is known to the scheduler it can also reduce inter-core contention by assigning stages that do not interact to separate cores and/or separate processors. Conversely, when two stages are known to interact heavily they can be scheduled on the same core or a neighboring core in order to increase data cache re-use. This is similar to a contractive mapping of processes to machines in a cluster [HD94].

1.5 Outline

The last decade has seen numerous advances in server software architectures, particularly in the way server logic is structured to handle many requests concurrently. The next chapter surveys the state of the art in staged and other server architectures. Chapter 3 presents the main contributions of this dissertation and places them in the context of previous work. Chapter 4 is the crux of the dissertation: in a set of experiments I evaluate the performance of staged concurrency and stage scheduling policies on representative servers. In chapter 5 I describe some of my experiences designing and implementing staged servers. Chapter 6 briefly surveys related work, and the text concludes with a chapter on future developments.

1.5.1 Results

The research reported in this dissertation starts from the basic premise that stages are a fundamentally superior way of handling concurrency compared to naive kernel thread-per-connection approaches. Rather than concentrating on the advantages of the concurrency model over other models, I have focused on variations within the same model, particularly scheduling policies for staged servers. In particular, I will show that:

1. Stage scheduling policies that multiplex stages over one kernel thread per core outperform policies that use more threads in terms of throughput on a CPU-intensive workload.

2. Thread-per-core policies with feedback-driven stage selection heuristics are comparable to those with simpler heuristics on the same benchmarks, despite expectations that the former policies should exhibit better performance – a negative result.

3. Load balancing can increase the throughput of staged servers scheduled with thread-per-core policies.

1.5.2 Scope

The primary focus of this dissertation is on optimizing CPU scheduling of staged servers to increase throughput. Other optimization targets, such as reducing power consumption

[MB08], scheduling to meet real-time deadlines [AB90], maintaining Quality of Service guarantees [LMB+96] and target I/O rates [MP89, SGG+99], and provisioning virtual machines [KC07], while equally valid, are outside the scope of the current work.

Furthermore, I have limited my investigations to CPU scheduling in userspace. Although some of the techniques proposed here could be adapted to kernel schedulers, I have not attempted to do so. Instead I have focused on user-level schedulers that multiplex stock kernel threads. A few of the algorithms presented in this dissertation require the ability to explicitly fix the processor affinity of a thread; a system call for this purpose is available on most server-class systems.

Finally, in the course of this work I will consider CPU scheduling on a single machine only. Coordinated CPU scheduling on multiple machines can improve the performance of distributed applications [ADCM98], but this is beyond the scope of my research.

1.6 Terminology

In writing this dissertation I have assumed that readers are familiar with the basic mechanisms, algorithms, data structures, and other artifacts of CPU scheduling as well as the terminology used to describe them (processes, run queues, quanta, etc.). For readers who are not familiar with these terms, I recommend Marshall McKusick and George Neville-Neil's *The Design and Implementation of the FreeBSD Operating System* [MNN04] as an introduction to CPU scheduling in operating systems. Before proceeding I would like to clarify my use of a few of these terms in order to avoid misunderstanding:

Processes and threads

I will use the term *thread* in contexts where the terms *process* (a.k.a. heavyweight processes) and *thread* are interchangeable, and I will consider threads as equivalent to lightweight processes that share an address space. In all cases the unmodified use of the nouns “process” or “thread” implies *kernel process* and *kernel thread*, respectively; *user-level threads* are referred to by name where appropriate. I will also prefer the term *user-level* to *application-level*, as in “user-level threads” or “user-level scheduler”, on the assumption that the two modifiers are synonymous in the context of this dissertation.

Core

The current generation of processors combine multiple *cores* in a single physical package on the processor die. The cores of a processor have individual L1 caches but usually share L2 and occasionally L3 instruction and data caches. From the perspective of the CPU scheduler a core is the unit of concurrency in hardware: two different cores can execute two different instruction streams simultaneously, rather than merely providing the appearance of concurrency while multiplexing at the hardware level, as in simultaneous multithreading. The distinction is an important one, since one of the goals of the stage scheduling algorithms discussed here is to minimize cache contention between cores, especially on L1 and L2 caches. In order to do so cores must be considered separately. Because a processor may consist of multiple cores and thus be able to execute multiple instructions simultaneously, I will prefer the term “core” to “processor” or “CPU” when individual cores are targeted. In accordance with accepted nomenclature I will use the term *multiprocessor* to refer to both multicore (two or more cores in a package) and traditional multiprocessor (cores in separate packages) setups. The general term "CPU" will be relegated to contexts where any or all cores and processors are involved, e.g. "CPU-bound".

CPU scheduler

Although the term "CPU scheduler" is normally written in the singular, in multiprocessor systems the CPU scheduler is usually not a monolithic entity that has global knowledge of system state and makes scheduling decisions for all cores, which would imply that one core makes scheduling decisions for itself and all others. Instead each core spends some of its cycles running an instance of a CPU scheduling algorithm, which is designed to interact with instances of the algorithm on different cores in a peer relationship. A limited set of in-memory data structures are shared between the instances of the algorithms, while each core also has its own private data structures. In the interests of simplicity I will refer to this system of interacting algorithms as the CPU scheduler, in the singular.

Commodity [operating systems] [kernels]

I will use this generic term to refer to the most commonly-used server operating systems as of this writing: Microsoft Windows and Linux, and, to a lesser extent, BSD derivatives and Sun Solaris. Because of their widespread deployments and user bases Windows and Linux are the primary targets of the current work.

Event

The term *event* is used in several contexts in this dissertation. Generally speaking, an event is equivalent to a message in a message passing system. The term can also refer to notifications that e.g. the kernel has read data from the network on a specific connection; the term *event-driven programming* is often used to describe a way of programming systems to "react" to events of this kind. For the purposes of this dissertation event-driven programming may be considered a specific case of message passing, where the sender is usually the kernel rather than a peer in the message passing system. The fact that the message may not actually contain the data is of secondary importance. I will use the term *event* in preference to "message" because the former term predominates in discussions of servers, while the latter is usually associated with more general distributed systems.

Stage

The term *stage* was inherited from previous researchers (see chapter 2). In the present context the term has two slightly different definitions, one conceptual and the other grounded in implementation. The conceptual definition of a stage is that of section 1.2 above. In implementation contexts a stage is a class of objects that consist of a event queue for communication between stages, an event handler (the concurrently-executable code), and any constructs for supporting a particular scheduling policy, such as locks around an event handler.

Clients and connections

Servers for stateless connection-oriented protocols such as HTTP and SMTP are usually designed to be agnostic of whether client software has one connection to the server or many in parallel. The most popular web browsers in use today open at least two connections to a given web server in order to increase request concurrency and decrease user-perceived latency, and browsers can be configured to make 4, 8, or more connections to a server[5]. In normal operation the server treats each connection separately, both in terms of concurrent processing and state maintenance. Thus I will prefer terms such as "thread-per-connection" and "connection state" to "thread-per-client" and "client state", since the former more accurately reflect the server's internal design.

[5]`http://www.die.net/musings/page_load_time/`

Chapter 2

Background

This chapter surveys the state of the art in server concurrency models and stage scheduling policies, using an HTTP file server as an illustrative example. Although HTTP file servers are not always CPU-bound or even CPU-intensive, historically they have been the proving ground for new server architectures and algorithms. In comparison to more complex servers the set of basic operations of an HTTP file server is also quite limited, which makes the server convenient as an example. These operations are:

1. Accept a TCP connection from the network
2. Read a request from the connection
3. Parse HTTP request headers
4. Translate the request URL into a file path
5. Read the file from disk
6. Generate HTTP response headers
7. Send the response headers over the network along with the file as the response body

A production server would also have facilities for logging, URL pattern matching, and the like, but these are secondary to the main operations of the server.

Concurrency models

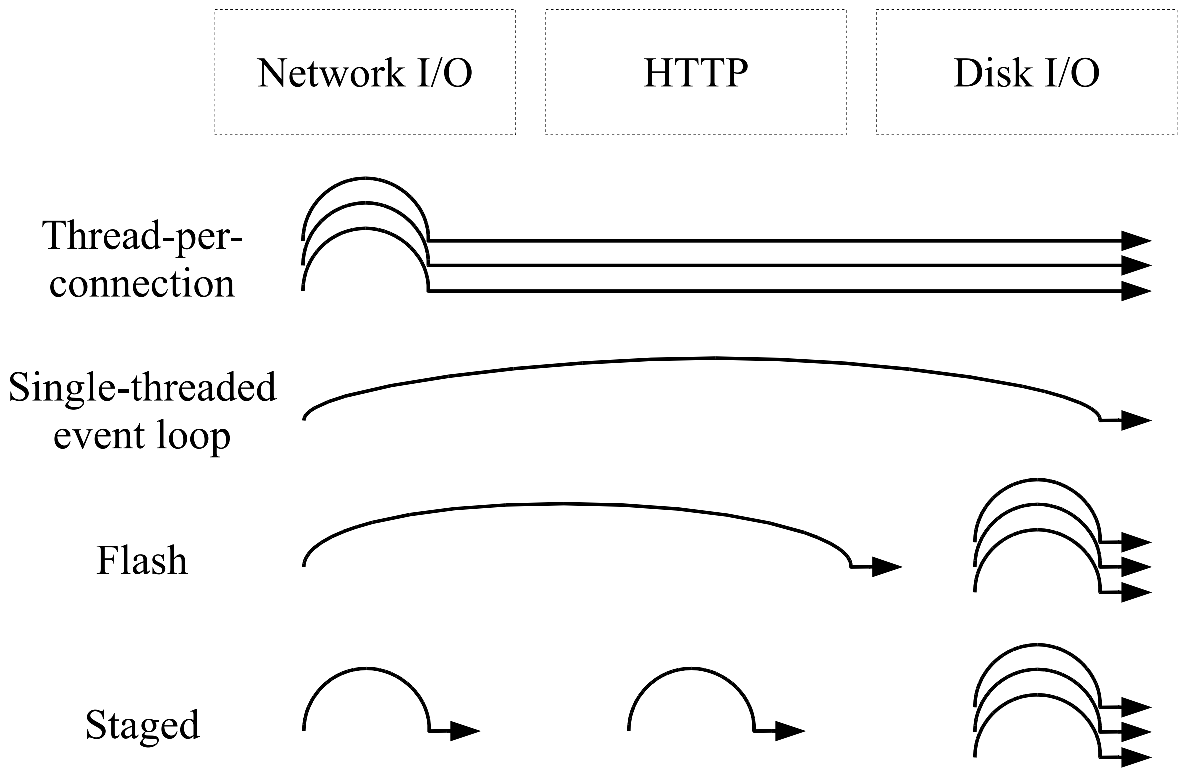
Server architectures can be classified according to numerous criteria, from the way servers utilize operating system I/O primitives to their mechanisms for dealing with overload. The *concurrency model* of a server architecture is one of its more definitive traits. A server's concurrency model describes how the server multiplexes connections, requests, and operations on requests using limited hardware resources.

Figure 2.1 illustrates four concurrency models for servers, roughly in the order in which they were developed. The basic operations of HTTP file servers, listed above, have been separated into three general areas – network I/O, HTTP handling, and disk I/O – which correspond to very coarse-grained stages in a stage concurrency model.

In the figure a continuous line with an arrow represents a single kernel thread and the progression of the line represents request processing over time. Lines that form a half circle, such as those at the beginning of the thread-per-connection row and the end of the Flash row, indicate that a kernel thread blocks waiting for some event to occur, such as an incoming connection or the completion of disk I/O.

The following sections describe each of the concurrency models in turn.

Figure 2.1: Concurrency models for HTTP file servers



2.1 Thread-per-connection concurrency

Kernel thread-per-connection concurrency is the basic model of the majority of HTTP servers, mail servers, database query engines, and other connection-oriented servers in production today. When a thread-per-connection server receives an incoming connection, the server creates a kernel thread or draws one from a bounded pool for the sole purpose of processing requests for that connection. In the case of the HTTP file server the thread executes operations 2-7 one after the other, reading from the connection with blocking system calls, parsing HTTP request headers, etc. with each thread executing the same sequence of operations in its entirety without interacting with other threads. The kernel assumes responsibility for time- and space-sharing cores, primary and secondary storage, and peripherals between threads so that all connections receive service. This is done transparently to the executing threads when they make blocking calls or run for an entire quantum and are preempted[1].

The primary advantage of kernel thread-per-connection concurrency is that it allows the programmer to implement request processing logic as a "straight" line of execution, which for the majority of developers is the most natural way of programming. Pushing all responsibility for multiplexing the underlying hardware into the kernel does come at a price, however. The problems with this concurrency model fall into several categories: read/write contention on shared resources; the effects of "excessive fairness"; the opacity of

[1]Further details on the mechanics of kernel schedulers can be found in [BC05] or [MNN04]. W. Richard Stevens's excellent *Unix Network Programming* includes a thorough introduction to thread-per-connection concurrency in servers.

threads from the perspective of the scheduler; the overhead of multiplexing many threads; and the problem of choosing the right number of kernel threads for a given workload. The following sections describe these issues in detail, along with some proposed solutions to them.

Read/write contention on shared resources

The thread-per-connection model of concurrency only works well when requests are largely independent of one another and the threads processing concurrent requests can be time-multiplexed by the kernel without the explicit direction of the programmer. Fortunately this is the case for some of the most well-studied servers, such as HTTP file servers, in which there is no read/write contention for the file system and other shared resources. Unfortunately it is not the case for less trivial daemons that read and write to secondary storage, such as SMTP servers. These servers must go through various contortions to avoid having a thread overwrite or interleave another thread's files [Ber95].

Writing multithreaded code that reads and writes shared data structures in memory is even more challenging, due in large part to the discipline and care required to use locks, the difficulty of reproducing concurrency-related bugs, and loose language integration [Boe05], among other factors [SL05]. Alternatives to the status quo in this area have been suggested (see e.g. [Lee06]), but these have made relatively little headway in practice.

Excessive fairness

In the past several mainstream operating system kernels such as those in Windows and Mac OS have scheduled threads cooperatively, assuming that threads would voluntarily relinquish access to the CPU to give other threads access. This assumption proved to be untenable in practice: inevitably some applications would hog the available resources and the performance of the system as a whole was degraded in the eyes of users. Today all major operating system schedulers are designed to assume by default that fairness must be enforced on threads using preemption. This is for the greater good of the system but it has had a markedly detrimental effect on the performance of servers.

For example, a thread in an HTTP file server may be a few statements away from writing a client's response to the network when the thread is preempted. The thread then has to wait for other threads to run; these other threads may or may not use all of their quanta, and the cycle time between successive runs of the same thread can be long and unpredictable. When the original thread finally receives a new quantum the data that it was going to write will have almost certainly been evicted from the cache due to the memory accesses of other threads, so the start of the quantum is wasted on processor stalls for lower-level caches and/or memory. Once the data is back in cache the thread executes a few statements and then enters the kernel again to write the data to the client's network socket. In the kernel the thread may have to block on space in the socket buffer, cutting off its quantum prematurely. The end effect is that the overhead of thread preemption and context switching can be significant in relation to the actual work accomplished. Unfortunately, the kernel scheduler has no sure way of knowing when a thread is "almost finished". Having that information would allow the scheduler to be more lenient and increase the quantum of finishing threads in order to decrease latency.

Lengthening the base quantum for all threads can reduce the likelihood of prematurely interrupting a thread and the deleterious cache effects that follow from it, but the duration of a thread's quantum is a tradeoff between responsiveness (shorter quanta) and throughput (longer quanta). By reducing the quantum a scheduler gives more threads a chance to run, but the effects of ill-timed preemption become more pronounced. Conversely, a longer quantum increases the likelihood that threads will run until completion (i.e. to a blocking point) but it also allows CPU-intensive threads to hog the CPU, which tends to increase user-perceived latency. Even when resource hogs are restricted [BDM98], the presence of many CPU-intensive threads will still mean longer inter-run times for threads that do not use their full quanta. This is alleviated somewhat by scheduling heuristics that prefer interactive (i.e. frequently blocking) threads, but in the end the kernel can only provide an upper bound on how long a given thread will have to wait for the CPU.

Opacity

The cache effects of fair thread scheduling are part of a more general problem for kernel thread schedulers, namely the opacity of thread logic in the view of the scheduler and the effects of opaque thread scheduling on CPU efficiency. Multiprocessor thread scheduling is another problem of this type. Most kernel schedulers use simple algorithms such as work stealing [BL94] and affinity scheduling [TTG93] to distribute and migrate threads between processors. These algorithms are usually agnostic of memory access patterns and concurrency between threads, so two threads working on similar data may run on two different cores, even when the system is not heavily loaded, which induces unnecessary inter-core cache traffic and stalls.

The opacity of memory access patterns and other thread behavior is partly the result of legacy API design, which traditionally gave preference to simplicity from the server programmer's viewpoint over other concerns, and thus necessitated the use of heuristics like "prefer interactive threads" that attempt to predict the behavior of a thread based on minimal information. Solutions to the problems that result from this opacity fall into two basic categories: those that maintain the conservative status quo of legacy APIs and attempt to extract more implicit information on thread behavior; and those that extend and supplement legacy APIs and encourage the programmer to provide more explicit information to the scheduler.

Extracting more implicit information

Many proposed improvements to existing kernel schedulers have relied on extracting side channel information on thread behavior rather than extending or supplementing legacy APIs. For example, several researchers have proposed scheduler optimizations that observe and predict the memory access patterns of threads [BP05, FSSN05, SMD07]. With knowledge of these patterns an algorithm can automatically schedule "compatible" threads on a single CPU or multiple CPUs in parallel or in sequence. The definition of compatible depends on the workload:

- If threads are known to be working cooperatively (e.g. in thread groups), then two threads are compatible if they read the same regions of memory and write to different regions, which implies an overall increase in the cache hit rate and decrease in inter-core cache coherency traffic in comparison to two threads that read different regions and write to the same region.

- However, for most workloads the algorithm designer must assume that threads are competitive, and the capacity/conflict misses from multiple threads may be unpredictable. Fedorova et al. [FSSN05] make the rather questionable assumption that cache misses will be uniformly distributed throughout a cache, so the cache miss rates of two competitive threads are linearly related.

Beyond this basic assumption the proposals also differ in their modes of operation:

- **Offline**: record memory accesses from a trial run of the application, construct a hypergraph with vertices = threads and edges = memory regions accessed by two threads, partition this graph across the available cores, and then run the application with the resulting static thread assignment [SMD07] or group related fine-grained threads on a core or processor [CGK+07].
- **Online**: determine a thread's "fair CPI [cycles per instruction] ratio" [FSSN05] or "performance ratio" [BP05] by observing hardware performance counters when the thread is running exclusively; find a compatible thread with similar performance characteristics; and compensate threads for extra cache misses / lower CPI induced by co-runners by increasing a thread's effective run time.

The offline approach proposed in [SMD07] is primarily intended for high-performance computing applications, where threads cooperate and their memory access patterns are deterministic across runs. The online proposals make a similar, though less explicit, assumption of consistent memory access patterns but use online sampling of exclusive thread execution to predict a thread's behavior under multiplexing conditions rather than entire offline runs with all threads executing. The two online optimizations [FSSN05, BP05] were evaluated on the SPECcpu2000 benchmark [Sta00]; it is not clear whether their improvements extend to other workloads such as the servers considered here. Predictive models and offline runs might be rendered unnecessary if the processor architecture can pinpoint cache sharing efficiently in real time, as in [TAS07], but most of today's commodity processors are not capable of this.

Changing legacy APIs

Some system researchers have been more optimistic about programmers' willingness to supply input and feedback to the kernel scheduler, and assumed that developers who want CPU efficiency will be willing to bear some inconvenience for it. Proposals in this vein vary according to how much input they ask of the programmer: from simply designating and describing groups of compatible threads [SPM07, RLA07] to application-level Quality of Service controllers [LMB+96]. Unfortunately, few research proposals of this kind have been integrated into commodity operating system kernels, which typically export a well-established but much more limited set of controls for users and developers to adjust in order to steer the scheduler from an application. These controls include:

- static and real-time priorities and "niceness", which affect the scheduler's long-term preference for one thread over others [MNN04]

- explicit proportions, which indicate what fraction of available CPU time a thread should receive in relation to other threads [WW94]
- real-time deadlines for thread execution [AB90]

While there are various means of translating one control to another (e.g. real-time deadlines into proportions [Reg01]), these tend to be coarse-grained at best, much like the controls themselves. Priority-based scheduling is especially ill-designed in this regard: because of priority decay, unexpected blocking, and the presence of other threads a thread's priority may not have a linear relationship to how much and how often the thread actually runs. Proportional share and real-time scheduling are slightly more dependable in this respect, but they require more precise input from the application, which may have to adapt this input continuously at run-time in reaction to changing load conditions. Even then the application still has no control over the order in which and duration for which threads are run in the short term, so the effects of detrimental context switching remain.

Overhead

Many of the most popular thread-per-connection servers in use today, such as the popular Apache web server [LL02], were originally designed to use one heavyweight process per connection and to create and destroy processes frequently. Over time the overhead of heavyweight process creation and destruction as well as their memory footprint spurred many developers to switch to lightweight processes or threads while retaining the same concurrency model, at the expense of some isolation.

Poorly-designed scheduling algorithms and synchronization primitives can be a serious problem when running many threads, especially on multiprocessors [ALL89]. Until recently many commodity kernels were only able to manage a few hundred runnable threads at a time without incurring significant runtime overhead [WCB01]. This was largely due to the use of $O(n)$ algorithms inherited from Unix variants for managing queues of runnable threads [MNN04]. The last five years have seen much improvement in this area, with the introduction of $O(1)$ algorithms and more accurate bookkeeping data structures [CCN$^+$05, Mol07] and hybrid userspace/kernel mutexes [FRK02] to the Linux kernel.

Choosing the right number of threads

The problems of contention, excessive fairness, and opacity are less pronounced when there are fewer threads in the system. With fewer threads in toto, each thread gets to run more often than it would have in the presence of more threads. However, a thread-per-connection server running with fewer threads can handle fewer connections. The server must find a balance between having enough threads to handle connections and limiting the effects of competition between threads, such as cache evictions and lock contention. Choosing the right number of kernel threads for a given workload, or even a minimum and maximum of that number, is a difficult problem, as evidenced by the abundance of web pages, books, and other sources offering advice on how to tune the most popular thread-per-connection servers [LL02, BD03, Lur06].

2.1.1 User-level schedulers

One solution to the problems of adverse kernel thread scheduling is to allow programs to observe and control their own CPU scheduling from userspace. A user-level scheduler can mitigate many of the problems described above by making more informed scheduling decisions. User-level schedulers also have the significant advantage of specialization. A user-level scheduler that has been designed, adapted, and tuned for a specific application or class of applications can almost always outperform a general kernel scheduler on the same applications. The kernel scheduler must be robust and reasonably efficient for many different kinds of workloads, but a user-level scheduler need not be.

The most direct way of scheduling in userspace is for the kernel to make an upcall into userspace whenever the kernel scheduler would normally make a scheduling decision, i.e. when a thread blocks or its quantum expires. This is the basic mechanism of scheduler activations [ABLL91]. The advantage of this approach is obvious: the user-level scheduler has full control over the kernel scheduler, which means it can ensure fairness (using preemption) and handle blocking gracefully. On the other hand, implementing scheduler activations efficiently is quite difficult, because the overhead of frequent kernel to userspace upcalls can be significant. Scheduler activations have been successfully implemented in the mainline FreeBSD[2], NetBSD [Wil02], and Solaris [Mau99] kernels and as patches to the Linux 2.4 kernel [DN03]. In these environments scheduler activations are a viable approach to fine-grained scheduling in userspace.

Not all operating systems support scheduler activations, however, and developers who wish to control the kernel schedulers in these operating systems (which include recent versions of Linux as well as all versions of Microsoft Windows) must do so through more indirect means, without explicit kernel cooperation:

1. By limiting the set of threads that are runnable at any one time to a subset of all threads in the system (usually one kernel thread per connection), in effect forcing the kernel to schedule only those threads

2. By reusing kernel threads (usually one thread per core) for different logical execution contexts (usually one context per connection)

The first approach is the modus operandi of several user-level schedulers, including ALPS [NP07], the Oracle Database Resource Manager [RCL01], and MS Manners [DB99]. This method has the advantages of simplicity and portability. The application can be designed to use conventional thread-per-connection concurrency, the scheduler has no extra abstractions to manipulate, and it can handle blocking fairly easily. The major disadvantages of this approach are essentially those of having one kernel thread per connection. For applications like the Oracle database, with only a few dozen or hundred inbound connections mainly executing I/O operations, the inefficiency associated with having many kernel threads does not dominate execution times, so this type of user-level scheduling is quite reasonable. For CPU-intensive servers that is much less likely.

The second approach to user-level scheduling is discussed in the next section on event-driven programming.

[2]`http://www.freebsd.org/kse/`

2.2 Single-threaded event loops

The single-threaded event loop was long perceived as the only viable alternative to thread-per-connection concurrency, and a remedy to the problems discussed in section 2.1. In classic event-driven servers like the Harvest web cache [CDN$^+$96] a single thread waits for events using an event notification system call such as `select` or `poll` and processes events in an endless loop. Connections are accepted, requests are read, and responses are written in non-blocking manner so that a single kernel thread can process requests from many connections. The endless event loop is occasionally referred to as an "event reactor" [Fet05] or the "reactor pattern" [SRSS00], because the code is structured as a set of reactions to certain system events (incoming connections, data ready to be read, data written). The code that "reacts" to an event is often called a *continuation*. Note that in the event loop the state for each connection must be saved before a blocking operation and restored before a continuation can run. This state is referred to as a *closure*, and the process of saving and restoring it in an event loop is called "manual stack management" [AHT$^+$02], as contrasted to the "automatic stack management" of threads, which automatically save and restore state before and after a blocking call.

The main problem with a single event loop is that not all blocking calls have non-blocking variants: to cite two examples, `open()`ing or `stat()`ing a file on disk may block if the file metadata is not in the operating system's cache, yet there are no portable non-blocking alternatives to these operations [RP04]. Because there is only a single kernel thread running the event loop a blocking `open()` or `stat()` halts the entire server until the blocking operation completes.

Now that multiprocessor machines have become the norm the use of a single thread is also a liability even when it doesn't block, because a single-threaded server can only exploit a fraction of a machine's cores. A multithreaded event loop requires the programmer or a compiler/preprocessor to carefully consider how to parcel events out to different cores in order to avoid data races. Schemes for accomplishing this include "coloring" events to indicate those can be processed in parallel (`libasync-smp` [ZYD$^+$03]) as well as using preprocessor macros to translate lock- and fork-like constructs to restrict and direct parallel event processing (Tame [KKK07]). Stage architectures are another, similar solution to the multicore event handling problem, and will be discussed below.

2.2.1 Threads vs. events

The relative merits of thread vs. event programming have been the subject of a long debate; see [Ous96] and [vBCB03] for opposing viewpoints. Event-driven programming is widely considered to be fundamentally more difficult than programming with threads. By preserving the [apparent] line of execution user-level threads facilitate a linear programming style that most developers understand more intuitively than continuations and closures. There is also much better tool and library support for thread programming, in large part because it is easier to extend single-threaded tools to account for multiple threads than it is to track an event-driven program. Proponents of thread programming argue that the performance penalties associated with threads – the penalties that pushed many serious developers to the event camp – can be remedied by better implementation (see section 2.1), while the difficulty of working with events is unlikely to recede[3].

[3]See [KKK07] for an attempt at rebutting this last argument.

Despite these differences, thread and event/continuation programming are conceptual duals of each other [LN79] and can co-exist in the same application [AHT$^+$02, LZ07]. Event loops are often used to emulate threads in userspace. A user-level threads library can abstract away the code for explicitly saving and restoring per-connection state and calling continuations, so the server programmer can write normal threaded code with "blocking" operations. When the server code calls one of these operations the user-level thread library substitutes code for saving and restoring thread state and inserts reentrant code to try a non-blocking variant of the operation. This form of automatic stack management can be implemented in several ways: at compile-time, with source- and bytecode-level transformations from automatic stack management to a manual equivalent [App92, DSVA06, FMM07, SM08]; at run-time within the operating system kernel [DBRD91] or as an OS-provided API like Windows fibers [Duf08]; or with user-level stack switching primitives such as the POSIX `getcontext` and `setcontext` calls [IEE88], signals and `longjmp`s [Eng00], or processor-specific assembly language.

In the last decade a number of hybrid systems have emerged that incorporate some of the strengths of thread programming (readable code paths, transparent multiplexing) while eschewing some of its weaknesses (locking, expensive context switching). Programming languages such as Erlang [Arm97] and user-level thread frameworks such as Kilim (for Java) [SM08] can multiplex tens of thousands of extremely lightweight threads. These systems use message passing between threads instead of locking in order to reduce contention and make multiprocessor thread scheduling simpler.

2.3 Flash

The Flash web server [PDZ99] was one of the first widely-publicized attempts by the academic community to reconcile the increasing demand placed on servers with existing process-oriented operating system APIs. Up until that point most researchers had focused on improving server software performance through conventional means: changing the operating system while preserving legacy interfaces and applications [BDM98]. These improvements rarely found their way into commodity operating systems, and thus had relatively little practical effect. The authors of Flash took the more pragmatic approach of adapting to the current state of commodity operating systems by working around some of their deficiencies.

Like many of its predecessors, Flash was optimized for serving static files from disk. Pai et al. distinguished Flash from existing web servers such as Apache by Flash's novel AMPED (Asynchronous Multiple Process Event Driven) concurrency model. AMPED was a hybrid of the classic single-threaded event loop and a worker thread pool for blocking on disk I/O. The Flash server executes all of the request processing steps (accepting, reading, et al.) in the event loop except blocking disk I/O and URI to file path translation. The design was dictated by the following condition: if a request could be satisfied immediately by the main server thread without blocking the main thread's event loop would execute all the steps necessary to respond to the request immediately. Otherwise the main thread would offload the blocking task to a worker thread, which would signal the main thread when the task was complete. The main server thread could then respond without blocking.

Flash combined the speed of the single-threaded event loop with the ability to block on legacy system calls for disk I/O. This was an extremely pragmatic strategy for its time.

In subsequent work the authors of Flash further optimized the server along the same lines: eliminating potential blocking paths at the kernel interface and below, improving file caching heuristics, and reordering requests to process shorter requests first [RP04].

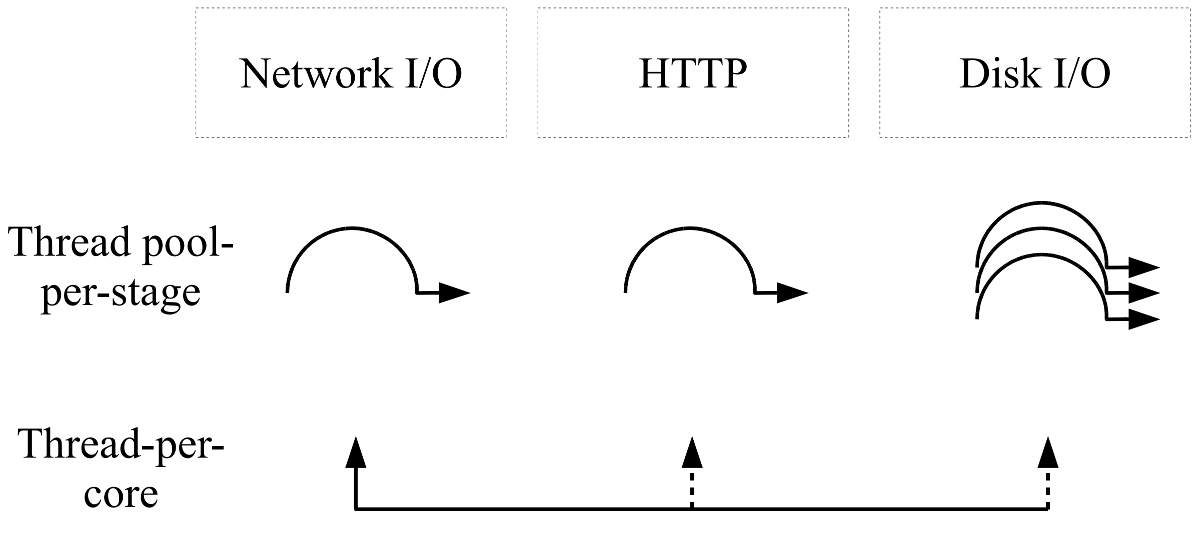
Despite its narrow target the Flash architecture was an important milestone in server design. It shifted the focus of server performance research from kernel to userspace, from improving operating system support for servers [BDM98] to designing servers that could better exploit existing systems [CM01a].

2.4 Stage concurrency

The staged server concurrency model introduced in chapter 1 was first publicly advocated in the late 1990s in a new breed of server architectures. These architectures shared a basic model of concurrently-executable code (stages) and mediating queues and were similar in many other respects. However, the architectures differed significantly in the the way they actually executed stages, i.e. their *stage scheduling policies*. A stage scheduling policy dictates how an application execute stages over time, in approximately the same way that a thread scheduling policy dictates how cores execute threads over time. Stage scheduling policies tend to be much less accurate and deterministic than thread scheduling policies, because stage scheduling is typically done in userspace, with cooperation between stages rather than preemption. There may also be more threads involved in a stage scheduling policy than there are cores in the system; time-sharing between these threads introduces additional non-determinism.

Figure 2.2 shows the two main classes of stage scheduling policies, thread pool-per-stage and thread-per-core. The lines and half circles have the same meaning as those of the concurrency models depicted in figure 2.1.

Figure 2.2: Stage scheduling policies



The following sections outline the historical development of these policies.

2.4.1 SEDA and the thread pool-per-stage policy

The Staged Event-Driven Architecture (SEDA) was the first widely-publicized stage architecture for servers [WCB01]. SEDA embodied a very specific notion of a stage architecture, one that was designed to support "well-conditioned services". Well-conditioned services are those that can degrade gracefully under overload by continuing to process requests with acceptable latency from as many connections as possible. Conventional thread-per-connection servers such as Apache are usually not well-conditioned: under heavy load the threads in these servers spend much of their time waiting on and competing for hardware resources (such as the CPU or the disk controller), which degrades service for all threads, leading to unacceptable latency for all connections. SEDA supported well-conditioned services by ensuring that servers spent as few hardware resources as possible on connections that cannot be served under a certain quality of service regime (such as a maximum response time) and refusing additional connections under overload conditions.

SEDA stages were defined by implementation rather than by an explicit notion of concurrency like the definition in chapter 1. A SEDA stage consisted of a signaled queue of events, an event handler, and a separate controller for admission control (of events) and feedback control of stage scheduling.

SEDA stages and the architecture as a whole were associated with a specific stage scheduling policy, one in which a pool of one or more threads is associated with each stage. Threads in the pool block on a stage's signaled event queue waiting for events, and a thread dequeues and processes events for only a single stage over the thread's lifetime. The major advantage of the thread pool-per-stage policy is that it allows a stage's event handler to block, perhaps unexpectedly. As noted previously, on some operating systems there are no nonblocking equivalents for some system calls, so having multiple threads executing a call can be a necessity.

The main challenge in implementing this policy lies in controlling the number of threads assigned to each stage's pool. In the original implementation of SEDA the number of threads in a stage's pool was increased or decreased by the stage's controller according to load conditions. For instance, if the length of a stage's event queue consistently exceeded a certain threshold the controller would increase the number of threads dequeueing and processing events for that stage, up to a certain limit on the thread pool size. In that implementation the controllers for each stage operated independently, which could cause severe problems with oscillation: two interacting stages would continually change their thread pool sizes in reaction to each other's behavior. Subsequent research by third parties has attempted to address the problems of tuning thread pool sizes [LLCZ06] and limiting the dilution of thread CPU time with a global cap on the number of threads [GR04]. User-level threads can also be substituted for kernel threads in the thread pool, which preserves the main advantage of the thread pool-per-stage policy – the ability to block in a stage – while allowing the user-level thread scheduler to control the order and duration of thread execution [HA05].

The disadvantages of thread pool-per-stage scheduling are similar to those of thread-per-connection concurrency: threads can be preempted at inopportune times, threads compete with one another, context switching overhead, etc. There is also the problem of single-threaded stages: adding and removing threads to a stage when only one thread can execute the stage safely is obviously not an effective way of increasing or decreasing the stage's output. Even for thread-safe stages, increasing and decreasing the number of threads at a stage is a very coarse-grained and unpredictable way of giving a stage more time on the CPU: depending on the blocking behavior of the stage's code, the number of threads at other stages, and the presence of intra-stage shared resources (including the event queue), the effects of adding a thread to a stage's thread pool can range from increasing throughput only slightly (e.g. when threads spend most of their time blocking on slow system calls, adding a single additional thread is unlikely to make much of a difference) to actually decreasing throughput by increasing contention for shared resources.

2.4.2 Thread-per-core policies

The thread pool-per-stage policy adds and removes threads to and from stages as a means of controlling CPU proportions and offsetting blocking, on the assumption that event handlers are likely to block and the server programmer will not work around this. For some servers, particular those that are CPU-bound, this assumption is too pessimistic and the overhead it induces unnecessary. More optimistic policies eliminate this overhead by only employing one thread per physical core of the machine, irrespective of the number of stages, in order to eliminate the problems that come with time-sharing threads on cores as well as to gain more control over the scheduler. The efficient use of a single thread per core is predicated on the assumption that stages will not block.

Cohort scheduling

Cohort scheduling was the first thread-per-core scheduling policy specifically designed for staged servers [LP02]. Like SEDA's thread pool-per-stage policy, it was an integral part of a larger platform for staged servers, called StagedServer. The StagedServer architecture and implementation were developed independently of SEDA in the late 1990s and published in 2002. Like SEDA, the StagedServer architecture featured explicitly-defined, event-driven stages with mediating queues and event handlers. It also utilized several of the same stage design patterns, such as partitioning a stage's data structures across multiple cores and pipelining between stages [WGBC00]. In addition to SEDA-like synchronized event queues, stages in a StagedServer-based application also incorporated per-core event stacks as a means of increasing data cache re-use. Both SEDA and the StagedServer architecture were designed to counter a specific problem with thread-per-connection concurrency, though it was not the same problem: while SEDA's well-conditioned services were an improvement on the poor overload behavior of thread-per-connection servers, the authors of [LP02] focused on the effects of thread context switching on data and instruction locality.

In a Cohort-scheduled server the stages of the server are *visited* by a set of kernel threads, one for each physical core of the machine, rather than having one or more kernel threads blocked on empty event queues, as in SEDA. Each thread/core in a Cohort-scheduled server iterates over the array of stages in a wavefront pattern, forward then backward, polling each stage's queue for events, processing any events in the queue by calling the event handler, then moving on to the next stage. If a thread repeatedly polls empty queues, it sleeps for exponentially increasing periods of time so the server can idle. The idea of visiting stages in a wavefront pattern was inspired by the observation that stage graphs usually resemble pipelines, with requests originating at the beginning of the pipeline and moving through subsequent stages and responses propagating along the reverse path. Linear pipelines such as this can be found in many staged servers, including simple HTTP file servers.

In normal operation a Cohort scheduling thread would drain a stage's event queue before moving on to the next stage. Under heavy load completely draining each stage's queue could lead to long inter-stage visit times, which would increase request processing latency. In order to avoid this situation the Cohort scheduling policy dictated that a thread/core should switch from one stage to another if the other stage's queue length exceeded a certain threshold or the other stage's inter-visit time exceeded a fixed interval. In either case the thread would finish processing an event at the stage it was currently visiting, then switch to visiting the threshold-exceeding stage until its queue was drained, then switch back to the original stage. If another stage exceeded its thresholds while a threshold-exceeding stage was being visited, the thread would not switch again, but continue processing the first threshold-exceeding stage, switch back to the original stage, and continue the wavefront as usual.

Other thread-per-core policies

Only one group has publicly attempted to apply the StagedServer techniques to other contexts, in this case a staged database management system (DBMS). In 2002 Harizopoulos et al. proposed several variations on Cohort scheduling [HA02]:

- `D-gated`: process all events in the queue in FCFS order, excluding those that arrived during the visit
- `T-gated(N)`: process all events in the queue in FCFS order, excluding those that arrived during the visit, up to a maximum batch size N
- `C-gated`: process all events in the queue in FCFS order, unless an event takes longer to process than a cutoff value, in which case subsequent events from the queue should be processed before returning to the large event(s) on the next visit – equivalent to Processor Sharing (PS) for large events and FCFS for small events

The first two regimes had previously been investigated by the authors of the Cohort scheduling paper, who reported that the size of batches was inversely proportional to response times on a latency-oriented benchmark: a batch size of four increased the mean response time of the server by 21%, while a batch size of twenty increased mean response time by only 9% [LP02]. In [HA02] Harizopolous et al. reported that the `C-gated` service regimes exhibited higher throughput than both `D-gated` and `T-gated(N)` regimes in simulations.

In later implementation work [HA05] Harizopolous adopted a hybrid approach between SEDA and Cohort scheduling, incorporating some aspects of both:

- Each stage is assigned a pool of user-level threads (vs. SEDA's kernel threads), which can "block" on I/O, ceding control to the user-level scheduler.
- There is a set of kernel threads, one per physical core, that visit stages.

- When a stage is visited the priority of its thread group is raised so that the off-the-shelf user-level thread scheduler will run the threads on the visiting core.
- A visit continues until a stage's queue is empty, one of the gate conditions above is reached, or all threads in the thread group have blocked on I/O.

This approach is well suited to a staged database, since it makes writing I/O-intensive stages easier – the user-level threads in a stage can simply block, instead of explicitly passing events to another stage – but without the overhead of SEDA's per-stage kernel thread pools. The gated policies (particularly `C-gated`) are also appropriate for systems in which processing times take on a heavy-tailed distribution, and it is necessary to prevent the head-of-line blocking that can occur in pure exhaustive FCFS systems such as Cohort scheduling. A Shortest Remaining Processing Time (SRPT) queue discipline might be a better answer to this problem, however.

Further details on Harizopolous's work on staged databases can be found in chapter 6 on related work.

2.5 Summary

In this chapter I have surveyed the state of the art in server concurrency models, from classic thread-per-connection and event-driven concurrency to staged concurrency to hybrid systems such as the Flash web server. I have particularly focused on the similarities and differences between servers based on the staged model, including SEDA and StagedServer as well as later refinements.

The next chapter will introduce the new technical contributions of my work on scheduling and load balancing in staged servers.

Chapter 3

Contributions

This chapter presents the main technical contributions of my work: four new thread-per-core stage scheduling policies and two strategies for balancing load across cores in thread-per-core-scheduled servers.

3.1 Improving thread-per-core stage scheduling

In developing and benchmarking the StagedServer architecture the authors of [LP02] were able to prove that staged servers combined with cache-conscious scheduling policies can offer significant performance advantages over conventional thread-per-connection servers. SEDA likewise proved that a staged server could more gracefully degrade its service under overload condition than a thread-per-connection server. There is significant evidence from these and other authors [PBH+07] that stage concurrency is fundamentally superior to more conventional concurrency models such as thread-per-connection concurrency.

However, despite these encouraging signs, there has been relatively little investigation into the design space of staged servers since the early 2000s. Stage scheduling policies are one of the more important dimensions of this space, especially the class of thread-per-core policies. Policies in this class share the basic mechanisms of Cohort scheduling: threads pinned to each core of a machine visit the available stages; threads idle when most of the queues in the system are empty; and there is no allowance for blocking in stages.

3.1.1 The MG1 policy

Over the course of my research I have experimented with numerous thread-per-core policies, though only a few could be considered useful. One of these was MG1. Like Cohort scheduling's wavefront pattern, the MG1 policy was based on a very simple idea, namely that stages should receive time on the CPU in proportion to their load. Here the load of a stage i is its queueing theoretic load:

$$\rho_i = \frac{\lambda_i}{\mu_i} = \frac{\text{arrival rate}_i}{\text{service rate}_i} = \frac{\text{the rate at which new events arrive at stage } i\text{'s queue}}{\text{the rate at which stage } i\text{'s event handler can process events}}$$

Visiting stages in proportion to their load has two purposes:

1. Reducing the number of visits to idle or mostly idle stages in comparison to "egalitarian" visiting patterns such as Cohort scheduling in order to reduce polling overhead (cache misses, atomic operations).

2. Ensuring that heavily-loaded stages are visited much more frequently than other stages. This partly obviates the need for Cohort scheduling's overflow/interval threshold at each stage, which served much the same purpose but in a less explicit manner.

The basic idea and formulas behind the MG1 policy were adapted from a 1993 paper by Boxma et al. entitled "Efficient Visit Orders for Polling Systems" [BLW93]. The next section briefly outlines the theory of polling systems before describing the MG1 policy in detail.

Polling systems

From a theoretical perspective, the staged servers described in this dissertation can all be modeled as a multiclass queueing system with k job classes (event types) and j single server stations (stages), each with a single queue in front of it. Given arrival and service rate distributions for the various job classes and stations, this model can accurately describe the runtime behavior of a server in steady state. The problem is that in order to reach a steady state the scheduling policy must be fixed, which is at cross-purposes with the primary goal of this dissertation: finding scheduling policies that lead to high-throughput steady states.

There are other formulations, derived from queueing networks, where the server:queue ratio does not have to be 1:1, and a single server can process jobs (events) from multiple queues. In these models a server is analogous to a physical core of a machine rather than a stage. *Polling systems* are the general form of 1:n server:queue models. In a polling system a single server (thread/core) polls a set of queues (stages) in some order, processing some or all of the jobs (events) in each queue before moving on to the next queue. Polling systems were introduced in the late 1950s to model a patrolling repairman, and have since been used to analyze a wide range of applications, notably token ring networks and other shared-medium scheduling problems[1]. The essential parameters of a polling system are listed in table 3.1.

The literature on polling systems largely consists of theoretical analyses, although some researchers have investigated real-world polling systems. To cite one example, Cheng et al. [CSO99] analyzed factories where a machine must switch between producing different types of goods. These factories can be modeled as polling systems with setup times, a.k.a. "multiclass production systems with setups". In [CSO99] Cheng derived a scheduling heuristic in which queues with the largest total scaled age are preferred. The total scaled age of a queue is the age of the jobs in a queue (i.e. how long they have been waiting in that queue) divided by expected service time. The authors of [CSO99] ran a numerical analysis of this heuristic on several industry data sets, and it compared favorably with some of the known heuristics for these systems, such as "always visit the queue with the most work". The essential problem with these and similar heuristics is that they tend to be tightly bound to a specific class of system. For example, Cheng et al. assume that no

[1]See [WWB07] for more background and references on scheduling in polling systems.

Table 3.1: Parameters of a polling system

Parameter	Typical values
visit order	**cyclic**: front-to-back, front-to-back **elevator**: front-to-back, back-to-front **dynamic or fixed Hamiltonian cycles** **other fixed orders**
service regime	**exhaustive**: process all jobs in the queue, including those that arrived during the visit **gated**: process all jobs in the queue, excluding those that arrived during the visit **n-gated**: process at most n jobs from the queue **time-gated**: process jobs from the queue until a time period has elapsed or the queue is empty
queue discipline	**First Come First Serve** (FCFS) **Shortest Remaining Processing Time** (SRPT) **Last Come First Serve** (LCFS)

setups are done at empty queues, i.e. the server/machine knows enough about the state of the system not to waste time polling empty queues. This is obviously not the case in the servers considered here.

Theoretical considerations aside, the Cohort scheduling policy can be described in terms of polling system parameters as a basis for comparison with other policies. A Cohort-scheduled server is a polling system with an elevator visit order; exhaustive or time/n-gated service, depending on system conditions (e.g. overloaded stages); and FCFS and LCFS queueing disciplines. The phased switching between polling system parameters is quite unorthodox and is rarely considered in the polling systems literature, though in light of the benchmark results of [LP02] it is hard to argue that this is a practical issue.

Boxma's efficient visit orders

The MG1 policy is slightly more orthodox in its design. It uses an exhaustive service regime, the FCFS queue discipline, and a polling table visit order. A polling table is simply an array that dictates the order in which stages should be visited. For example, the polling table $0, 1, 2, 0$ means that a thread should visit stage 0, then stage 1, then stage 2, and finally stage 0. The polling tables for the MG1 policy are constructed in such a way that the most heavily-loaded stages are visited more frequently than mostly-idle stages. This is Boxma's algorithm for efficient visit orders. The frequency with which stage i is visited with exhaustive service is calculated as:

$$f_i = \frac{\sqrt{\rho_i(1-\rho_i)}}{\sum_j \sqrt{\rho_j(1-\rho_j)}}$$

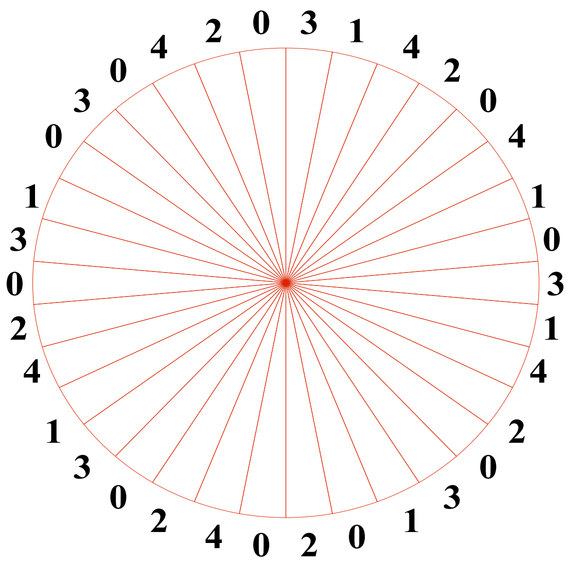
These frequencies are each multiplied by the size of the polling table M such that $Mf_1, ..., Mf_n$ are as close to integers as possible. The resulting values $m_1, ..., m_n$ are the total number of times each stage is visited in that polling table. The m_i visits for each stage are distributed throughout the polling table using the following procedure from [BLW93]:

> Let $\phi^{-1} = \frac{1}{2}(\sqrt{5}-1) = 0.618034...$. (ϕ^{-1} is also known as the Golden Ratio; it is related to the Fibonacci numbers $F_1, F_2, ..., F_n$ via $F_k = [\phi^k - (1-\phi)^k/\sqrt{5}]$.) Put the M numbers $\phi^{-1} mod 1, 2\phi^{-1} mod 1, ..., M\phi^{-1} mod 1$ in increasing order. (This corresponds to placing them on a circle of unit circumference.) Let the jth smallest number correspond to the jth position in the table. Assign $\phi^{-1} mod 1, 2\phi^{-1} mod 1, ..., m_1\phi^{-1} mod 1$ to Q_1 [stage 1], $(m_1+1)\phi^{-1} mod 1, ..., (m_1+m_2)\phi^{-}2 mod 1$ to Q_2 [stage 2], etc.

Note that $mod 1$ simply distributes the numbers in the golden circle between 0 and 1 so that each can be considered a percentage of the total number of polling entries in the table.

This algorithm ensures that there are at most three different interval lengths between successive placements (visits) of a stage. If M is a Fibonacci number there are at most two different interval lengths. Successive visits to a stage are evenly distributed while maintaining the property that stage i is visited with frequency f_i. Figure 3.1 illustrates a polling table with 34 entries (34 is a Fibonacci number) for a server with five stages.

Figure 3.1: An example polling table



In this example the relative queueing theoretic loads of the stages are approximately the same (close to 0.50), so the stages visits occur with approximately the same frequency, with stage 0 receiving 10 visits at evenly-spaced intervals and the other stages receiving 6 visits apiece (The slight imbalance in favor of stage 0 is an artifact of the small polling table size, necessary for a simpler illustration, due to to $f_i * M$ being rounded down to the nearest integer array index. In practice the algorithm uses polling table sizes of 144 or greater in order to ameliorate such imbalances).

3.1.2 The SRPT policy

The implementations of the Cohort and MG1 scheduling policies share much of their code base: one thread per core, conditional locking on single-threaded stages, exponential idling, etc. The main difference between the two is the way the next stage to visit is chosen by each thread/core: a wavefront in Cohort scheduling, polling tables in MG1. Isolating this difference in the implementation made it relatively straightforward to experiment with other visit patterns. One of these was the SRPT scheduling policy, a variation on Cohort's wavefront heuristic for staged servers with pipelines. Instead of visiting stages in a wavefront, SRPT always prefers stages at the end of the pipeline. These stages are visited first, and a core works its way back to the head of the pipeline. As soon as any stage is successfully visited (i.e. at least one event was processed), the scheduler restarts at the end of the pipeline. For compact stage graphs this very roughly approximates the Shortest Remaining Processing Time queue discipline, which is known to minimize response times and maximize throughput for most workloads. The implementation of this policy is shown below.

```
class SRPTVisitPolicy : public VisitPolicy
{
public:
  SRPTVisitPolicy( Stage** stages ) : VisitPolicy( stages )
  {
    next_stage_i = 0;
  }

  // VisitPolicy
  inline Stage*
    getNextStageToVisit( bool last_visit_was_successful )
  {
    if ( last_visit_was_successful )
      next_stage_i = 0;
    else
      next_stage_i = ( next_stage_i + 1 ) %
                     YIELD_STAGES_PER_GROUP_MAX;

    return stages[next_stage_i];
  }

private:
  unsigned char next_stage_i;
};
```

3.1.3 The DBR policy

The Drum Buffer Rope (DBR) stage scheduling policy was loosely inspired by Goldratt's Theory of Constraints [Gol90], the essence of which can be boiled down to "schedule everything around the bottleneck." The DBR policy prefers stages in ascending order of their service rates, so that slower stages are visited frequently. Like the SRPT policy, the DBR restarts polling at the first (slowest) stage after any stage has been successfully visited. The implementation of this policy is shown below.

```
class DBRVisitPolicy : public VisitPolicy
{
public:
   DBRVisitPolicy( Stage** stages )
    : VisitPolicy( stages )
  {
    next_stage_i = YIELD_STAGES_PER_GROUP_MAX;
    memset( sorted_stages, 0, sizeof( sorted_stages ) );
  }

  // VisitPolicy
  inline Stage*
    getNextStageToVisit( bool last_visit_was_successful )
  {
    if ( last_visit_was_successful )
    {
      next_stage_i = 0;
      return sorted_stages[0];
    }
    else if ( next_stage_i < YIELD_STAGES_PER_GROUP_MAX )
      return sorted_stages[next_stage_i++];
    else
    {
      memcpy_s( sorted_stages,
                sizeof( sorted_stages ),
                stages,
                sizeof( sorted_stages ) );
      std::sort( &sorted_stages[0],
                 &sorted_stages[YIELD_STAGES_PER_GROUP_MAX-1],
                 compare_stages() );
      next_stage_i = 0;
      return sorted_stages[0];
    }
  }

private:
  uint8_t next_stage_i;
  Stage* sorted_stages[YIELD_STAGES_PER_GROUP_MAX];
  struct compare_stages
    : public std::binary_function<Stage*, Stage*, bool>
  {
    bool operator()( Stage* left, Stage* right )
    {
      if ( left != NULL )
      {
        if ( right != NULL )
          return left->get_service_rate_s() <
                 right->get_service_rate_s();
        else
          return true;
      }
      else
        return false;
    }
  };
};
```

3.1.4 The Color policy

While the MG1, SRPT, and DBR policies inherited the basic mechanisms of Cohort scheduling (polling, idling, etc.), the Color policy is a slightly different take on thread-per-core stage scheduling. Rather than polling stages for new events as in Cohort scheduling, each thread/core in a Color-scheduled system waits on a central event queue, shared between stages. Each time an event is enqueued it is associated with a specific stage, which is then visited when the event is dequeued. This is akin to Zeldovich's event "coloring" [ZYD$^+$03] to indicate parallelism, with stages representing different colors. Like thread pool-per-stage policies, the Color policy has an advantage over polling thread-per-core policies in that it never visits an idle stage. Unlike thread pool-per-stage scheduling, however, the Color policy never uses more than one thread per core, so it also retains the main advantages (instruction, data locality) of thread-per-core policies. These advantages offset the cost of having a central point of contention, the shared event queue.

3.2 Load balancing for thread-per-core policies

The thread-per-core policies discussed thus far were designed on the assumption that every thread/core in the system visits all of the available stages and there is only one instance of each stage in the system. This design is appropriate for most servers. However, there are also situations in which explicitly segregating stages on different cores, e.g. load balancing, can lead to better performance by reducing lock contention on single-threaded stages and increasing cache hits by keeping instructions and data local to a core. This section discusses two load balancing strategies for thread-per-core-scheduled servers.

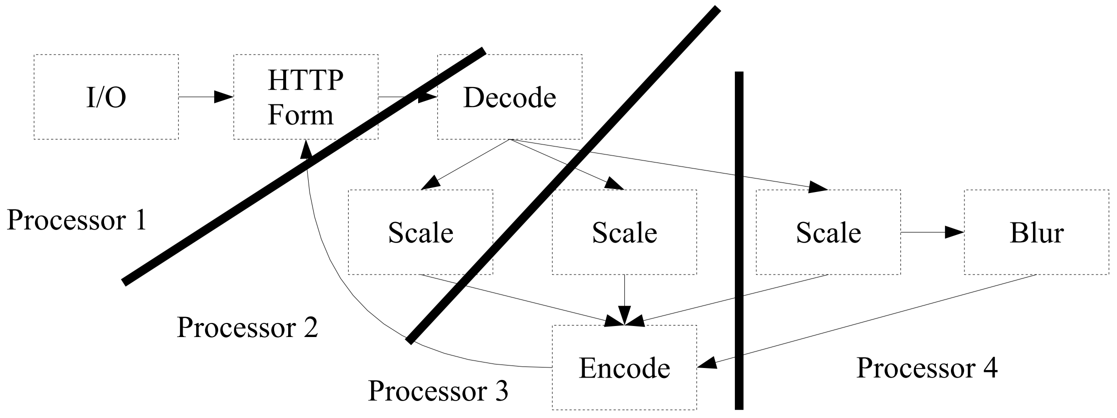
3.2.1 Replicating the stage graph

The first load balancing strategy is simply to replicate the entire stage graph on each core. A single I/O front end distributes incoming requests to the different replicas using a simple algorithm such as round robin. The main advantage of this strategy is that it approaches the ideal of Cohort scheduling, with each processor forming batches (cohorts) of requests that proceed through the stages together. Data cache contention is minimal because request-related data is local to a single processor. Likewise, there is no lock contention on single-threaded stages such as caches, since there is a replica of each single-threaded stage on every processor. This can obviously be a disadvantage as well: if caching is crucial to the performance of the server, replicating the cache on several processors makes cache hits less likely, and can offset some of the other gains of this strategy.

3.2.2 Partitioning the stage graph

The second load balancing strategy is to divide the stage graph into p partitions, where p is usually some divisor of the number of cores. A partitioned stage graph is shown in figure 3.2.

Figure 3.2: Partitioned image processing server



The goal of partitioning a stage graph is to balance the load assigned to each partition while minimizing the *edgecut* between partitions of a graph. The load of a partition is defined as the sum of the vertex weights in that partition, while the edgecut is the sum of the weights of the edges that cross between two partitions. For stage graphs the weight of a stage/vertex is defined as the load of that stage relative to other stages, as observed in the total amount of time spent processing events at a given stage over a relatively long period. In defining edge weights I assume that all events are of equal importance and simply count the number of events that pass between two stages over the same period.

Practically speaking, partitioning the stage graph effectively means keeping all of the cores of a machine equally busy; increasing instruction cache hits by limiting the number of stages visited by a given core; having only a single instance of a cache in the system; and minimizing inter-core data cache contention.

There are numerous algorithms for partitioning weighted graphs, such as the Kernighan-Lin [KL70] algorithm and its descendants. For the experiments in chapter 4 I measured the load and communications patterns of stages in the benchmark server in a trial run, then used the METIS family of algorithms [SKK00] to partition the weighted stage graph offline. The server was then restarted and benchmarked with the stages in a given partition pinned to one or more cores (the number of cores divided by the number of partitions). The process of measuring and partitioning could also be done online at the expense of some additional code complexity, which, as will be seen, would not have been justified by the experiment results of chapter 4.

3.3 Summary

This chapter introduced the main technical contributions of my work: four new thread-per-core stage scheduling policies (MG1, SRPT, DBR, and Color) and two strategies for balancing load across cores in thread-per-core-scheduled servers (partitioning and replication). The new policies are similar in their basic aspects to Larus and Parkes's Cohort/wavefront scheduling algorithm. They differ in the way they select stages to visit, from feedback-driven heuristics (MG1) to simple fixed orders (SRPT).

In the next chapter I will evaluate the performance of the new stage scheduling policies in comparison to the state of the art in thread-per-core and thread pool-per-stage policies. I will also evaluate the effectiveness of the two load balancing strategies in increasing throughput.

Chapter 4

Evaluation

This chapter briefly describes Yield, a platform for implementing staged servers in C++, and presents empirical evaluations of different concurrency models and stage scheduling policies using two Yield-based servers.

4.1 Yield

Yield[1] is a stage architecture and a platform for staged applications. Yield is unique among stage architectures in that it is not tied to any one stage scheduling policy, but supports all of those listed in chapters 2 and 3 as well as thread-per-connection concurrency. Yield can host almost any kind of staged, event-driven application, such as user interfaces or software routers [KMC$^+$00], in contrast to Flash, Apache, and other servers that are specialized for specific protocols or applications of those protocols.

The development of Yield began in 2002. The code base was initially an adaptation of Welsh's SEDA implementation from Java into C++. This first C++ implementation was written by Felix Hupfeld as part of his dissertation work on reconciliation algorithms for distributed key-value databases [HG06]. In mid-2003 I joined Felix's project as a student assistant and assumed responsibility for the code that became Yield. Over the next two years I re-designed and re-implemented the core code base several times, eventually specializing Yield as a web application server. For my Master's thesis in early 2005 [Gor05] I benchmarked a Yield-based static file web server using the SEDA and thread-per-connection concurrency models in order to show that stage concurrency models like SEDA perform better and allow more accurate control of a server than conventional thread-per-connection concurrency. I also demonstrated that my server was competitive with Welsh's original SEDA implementation in Java. The focus of those experiments was on admission control, particularly token-based schemes for dynamically limiting the number of connections accepted by the server. The focus of the experiments in this chapter, in contrast, is on scheduling under normal load rather than overload.

After joining the Computer Laboratory in Cambridge in late 2005 I decided to take up staged servers again and try to rectify some of the shortcomings I perceived in existing stage architectures. I subsequently re-generalized Yield to support stage groups and event

[1]`http://yield.sf.net/`

targets (see below) and multiple inter-process communication protocols (HTTP, ONC-RPC, JSON-RPC) and significantly extended the platform library. The experimental servers evaluated in this chapter and the thread-per-core scheduling policies and load balancing strategies described in chapter 3 were designed and implemented between 2005 and 2008.

4.1.1 Architecture

The Yield architecture inherited the basic building blocks of SEDA and the Staged-Server architecture, namely events, event handlers, event queues, and stages. A Yield-based application consists of an explicitly-defined network of stages that communicate via reference-counted events. A Yield stage consists of an event queue and an event handler supplied by the programmer. Unlike SEDA and the StagedServer architecture, the Yield architecture also incorporates the notion of a *stage group*. Every stage is attached to a stage group, which is responsible for scheduling the stages attached to it. A stage group implements a specific stage scheduling policy, such as thread pool-per-stage or Cohort scheduling. Different stage groups with the same or different scheduling policies may co-exist within an application, although in practice there is usually only one stage group per application.

4.1.2 Implementation

Yield's C++ code base is modularized into three layers: a platform library; a stage framework; and a library for inter-process communication.

Platform library

The Yield platform library is a portable (Windows, Linux, FreeBSD, Mac OS X, Solaris) collection of objects that wrap platform-specific code. It was originally developed by Felix Hupfeld and mostly rewritten by the present author. The library includes platform-specific implementations of synchronization primitives (mutexes, semaphores, atomic operations), file system interfaces (memory-mapped files, stat, Unicode disk path representations, etc.), queues for kernel I/O notifications (`epoll`, `kqueue`, `poll`, `libaio`, Windows I/O completion ports), and utility classes for timers, logging, and the like.

Stage framework

The Yield stage framework contains interfaces and implementations of `Events`, `Stages`, `EventHandlers`, `EventQueues`, and `StageGroups`. Although the framework is similar in its essential design to the implementations of SEDA and the StagedServer architecture, the implementation diverges in a number of respects, including its conservative use of stages and its substitution of non-blocking event queues for blocking, mutex-protected equivalents.

Event targets

Yield's stage framework includes an interface for *event targets*. Event targets are objects that can receive events. A stage is an event target that receives events via a mediating event queue and processes events asynchronously by calling an event handler. Other event targets simply pass through an event that is "sent" to them directly to the event handler, so the event sender calls the target's event handling code. This pass-through event target is the default in Yield. Stages are reserved for CPU-intensive event handlers, where the overhead of queueing and crossing thread boundaries is outweighed by gains in batching, parallelism, and prioritization. Thread-per-core scheduling policies such as Cohort scheduling are also more efficient when there are fewer but more heavily-loaded stages to poll: the scheduler spends less time polling inactive or mostly-inactive stages and more time processing large batches of events at active stages.

Non-blocking event queues

The default event queue in Yield is a non-blocking finite queue [MS96], in contrast to SEDA's lock-protected `Vector`s in Java. The use of non-blocking queues reduces contention between threads dequeueing from the same structure and eliminates memory allocations of bookkeeping data, which can also block in the presence of contention.

Unlike blocking queues, however, the non-blocking queue comes with no built-in means of synchronization. Instead a semaphore-based signaling protocol allows consumers to sleep on an empty queue until they are notified by producers [in a separate thread] of a newly-enqueued event. The signaling mechanism can be optimized based on knowledge of the number of producer and consumer threads: for example, if there is only one consumer thread and it is known to be running, the producer need not signal after an enqueue operation [Hup02, MP89].

Interprocess communication

Yield can transport events over TCP, SSL, and UDP. Requests and responses in application-level protocols like HTTP are implemented as events that can deserialize and serialize themselves to socket connections. These connections are accepted, read from, and written to via asynchronous I/O (AIO) interfaces, which use native socket AIO where it is available (Windows) and simulate it with non-blocking I/O otherwise (Linux and Solaris, among others). Network I/O is threaded independently of the stage framework, with one thread per core processing completed network operations and making callbacks.

Although I have used Yield primarily as an application server, it is capable of acting both as a server and a client. This is needed to support any sort of distributed server application, such as distributed databases.

4.2 Experiment: comparing concurrency models and stage scheduling policies in an image processing server

The image processing server introduced in section 1.3 was the focus of my first experiment. The experiment was designed to compare different concurrency models (thread-per-connection, staged) and stage scheduling policies (thread pool-per-stage, thread-per-core) on a single Yield-based application.

4.2.1 Workload

As noted in chapter 1, the decode-scale-blur-encode image processing server was adapted from a batch processing script used by a real web site, `wie-ich.de`. The data set for this experiment consists of 947 megabytes of real image data in 4697 files uploaded by users of the site over a 20 hour period, from 9 AM to 5 AM the next day. Table 4.1 shows some statistics on the sizes of files in the data set.

Table 4.1: Image file size statistics

Minimum	398 bytes
Maximum	4.2 megabytes
Mean	204.4 kilobytes
Median	37.2 kilobytes
90th percentile	623.2 kilobytes

The statistics indicate that the majority of files are relatively small, less than 50 kilobytes, but there is a heavy tail of much larger files that pulls the mean well above the median. This is consistent with SPECweb99-like file size distributions as well as other researchers' analyses of file sizes in real systems [Nah02].

In addition to the data set, a realistic arrival rate is required for a good workload, and here there is a problem. Simply replaying the production server's request stream (i.e. duplicating its observed arrival rates) is unlikely to stress a machine that is even half as powerful as the production server, because the site's administrator has intentionally overprovisioned the hardware so that the server can comfortably handle normal load while using less than 50-80% of the CPU and other resources. This leaves enough headroom to accommodate occasional periods of heavy load and reduces the potential for embarrassing and costly overload. Most production services are over-provisioned this way: when a machine sees more than 50-80% utilization it is time to install another machine to take over part of the load. This means individual machines in a well-provisioned environment are rarely pushed to peak performance. Single node performance still matters in some cases, however: small operators may not be able to afford new machines, or the server software may have been implemented in such a way that sharing load between two machines is not a viable option. Even in an overprovisioned environment, software that makes efficient use of the hardware can delay the point at which new hardware is needed, possibly indefinitely.

With this in mind I intentionally ignored the arrival rates of the real request stream, choosing instead to send requests as quickly as possible in order to push the server to peak performance (but not into overload). This also had the desired effect of differentiating the performance of the stage scheduling algorithms to a greater degree: when the server is running idle most of the time there is plenty of room for inefficiency, so the algorithms are less distinguishable.

4.2.2 Client software

The client software for this benchmark was a simple multithreaded Python script. The main thread of this script read the entire data set into memory, pre-created the HTTP requests for uploading each image (one image per request), and pre-filled a queue with the set of requests. The main thread then created a set of worker threads to send the requests and image data to the server. Each worker thread initiated a single connection to the server, then proceeded to dequeue a request from the global queue, write the pre-created request string to the network, and read the server's response, repeating this cycle until the queue was empty. A queue was chosen in preference to dividing the set of requests into subsets for each worker node so that the stream of requests from different threads would closely resemble the real stream, albeit with far smaller intervals between requests.

During a benchmark run each worker thread would send requests as quickly as possible and keep track of response times. The server's response time was defined as the time between the completion of the client's `send`ing the request to the network and the completion of the client's `recv` of the response. Normally a response time does not include the time the client takes to read the response, but only the first byte of it, so there is no bias toward smaller responses. However, since responses from the image processing server consisted solely of a status line and HTTP headers the size of all responses was the same, and the difference between the time the first byte was read and the time the response was completely read was negligible relative to the time the server needed to process the request.

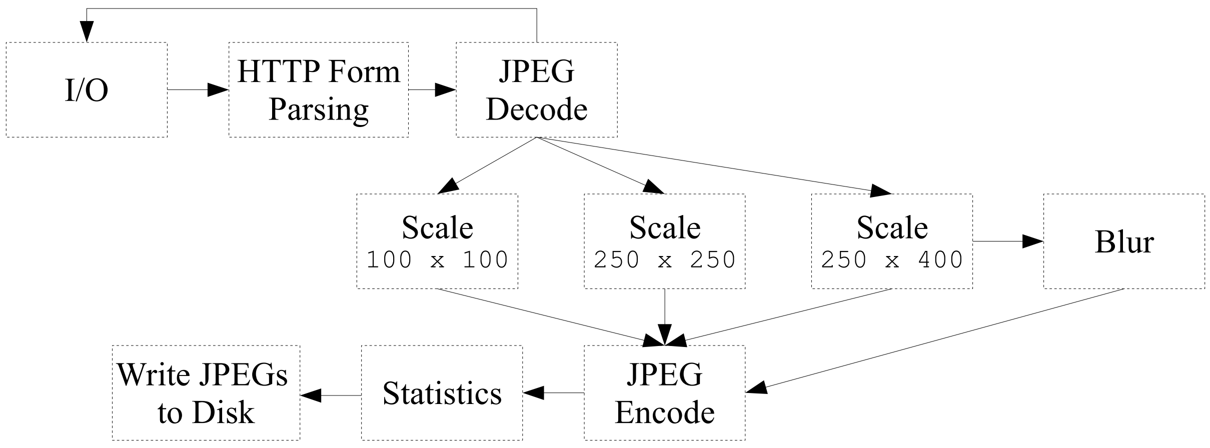
When there were no more requests to dequeue from the global queue a worker thread would signal the main thread that it was finished and then exit. Once all worker threads had finished the main thread would record a timestamp for the end of the benchmark run. The difference between this timestamp and the time when the worker threads were started was the duration of the experiment, which I referred to as the *client makespan*. The client makespan indicates how long the server took to successfully respond to all of the requests in the benchmark run. It is not the same as the server's makespan (i.e. the time the server needed to process every image from start to finish) because the server was designed to send HTTP responses to an uploaded image as soon as the image had been decoded (the stage where most errors occur) in order to minimize user-perceived latency while still reporting most errors.

One major caveat with this setup is the ratio of requests to connections. The client software utilized 4-16 worker threads, each of which initiated a single connection to the server, in order to upload 4697 images in total, so the ratio of requests to connections was 1174 to 1. This is hardly realistic in comparison to SPECweb99's 5-15 requests per connection, which was derived from analyses of real sessions. The limited number of client worker threads was chosen to match the number of threads in a kernel thread-per-connection version of the server (described in the next section), in order to demonstrate the effects of increasing the number of threads above the number of physical cores. More realistic connection patterns would have created additional I/O-related work on the server, but the cost of accepting more connections would still be dwarfed by the CPU consumption of the image processing stages.

4.2.3 Server software

The image processing server consists of a series of Yield stages for the different image transforms (decode, scale, blur, encode) as well as the support functions of the server (reading and parsing HTTP requests, collecting statistics, etc.). The stages are divided into two stage groups: the main group of non-blocking stages and an auxiliary group that contains a disk I/O stage for writing statistics and the output JPEG images to the file system. Figure 4.1 depicts the stages of the image processing server. This is the same figure as figure 1.1 in chapter 1.

Figure 4.1: Image processing server



Note that the “join” in the graph is not a real join in the classic sense, because the JPEG encoding stage does not wait for all of the versions of an image to arrive before processing any of them, but encodes each version as it arrives and forwards it to subsequent stages, asynchronously from other versions of the same image.

The image transform stages relied on two third party libraries, `libjpeg`[2] and CImg[3]. The `libjpeg` library provided an API for decoding and encoding JPEG images into a bitmap form CImg could manipulate. Both libraries are robust and perform well on current hardware. An attempt to use the more well-known ImageMagick[4] resulted in severe bottlenecks related to temporary file I/O and unnecessarily helpful locking on the part of the library.

Stage scheduling policies

Four stage scheduling policies were compared in this experiment: SEDA, Cohort scheduling, MG1, and SRPT. The Cohort scheduling algorithm was benchmarked in two configurations, one with per-core event stacks in addition to inter-core event queues and one

[2] `http://www.ijg.org/`

[3] `http://cimg.sourceforge.net/`

[4] `http://www.imagemagick.org/`

without; see appendix section A.1 for a discussion of these stacks and queues. SEDA-style thread pool-per-stage scheduling was tested with several thread pool configurations, shown in table 4.2.

Table 4.2: Image processing: SEDA configurations

Configuration	I/O threads	Form parser threads	Decoder threads	Blurrer threads	Encoder threads
SEDA	1	1	3	2	3
SEDA overthreaded	1	1	8	2	3
SEDA underthreaded	1	1	1	1	1

The original Java implementation of SEDA dynamically adjusted the number of threads at each stage in response to load. This is possible with Yield, but my experience has been that it often leads to oscillation of thread pool sizes. For this reason I chose to forego experimenting with dynamic thread pool resizing here; the configurations in table 4.2 were selected after a trial-and-error search of the space for representative configurations.

The thread-per-core policies (Cohort, MG1, SRPT) each employed a full complement of 4 threads to poll the five stages in the system. With the exception of the I/O event handler all of the event handlers in the system were thread-safe, so multiple threads/cores could execute the same event handler concurrently.

Thread-per-connection front end

For this experiment I also implemented a special kernel thread-per-connection front end. The front end was responsible for `accept`ing incoming connections and farming them off to worker threads, which then read HTTP requests and progressed through the event handlers of the image processing server as if the handlers were simply function calls. In other words, each of the worker threads executed the code for every event handler, from decoding to sending an HTTP response to scaling and encoding, before reading another HTTP request from the network. This configuration transformed the server into the functional equivalent of an Apache web server with back end modules for image processing.

Before each benchmark run, the pool of worker threads in this thread-per-connection server was pre-sized to match the number of worker threads on the client. A real thread-per-connection server would dynamically adjust the size of the worker thread pool in order to decrease the server's memory footprint in periods of low demand and increase the number of clients the server could handle concurrently in periods of high demand. However, the purpose of including the thread-per-connection front end in this experiment was simply to demonstrate that adding more worker threads to the server induces contention and competition between threads, which increases CPU use while decreasing throughput. Three different worker thread pool sizes were tested: 4, 8, and 16 threads, corresponding to 1, 2, and 4 times the number of physical cores in the machine.

Hoard

Some SEDA and thread-per-connection server configurations also incorporated Hoard [BMBW00], a `malloc` replacement for multithreaded servers. Contention on memory allocation is known to be a significant problem for multithreaded servers running on multiprocessor machines [MEG03]. Hoard tries to reduce contention on the global memory pool by having separate memory pools for each thread that fall back to the main pool when they are empty and release memory to the main pool when memory is scarce or some memory is not needed by a thread. Hoard relies on the heuristic that the thread that allocates a block of memory will also deallocate it, which is mostly true for a thread-per-connection server but not necessarily so for a staged server. For example, HTTP request data structures are usually allocated and deallocated in the same place, but the data structure used to hold images is allocated by one stage and deallocated by another. Depending on the number of cores in the system (> 1) and the stage scheduling policy, the latter situation may involve two or more threads, which means there may be contention on deallocation even if there is none on allocation. In these cases Hoard was excluded from the experiment setup.

4.2.4 Metrics

The primary metric of the image processing server evaluation is the number of bytes the server processes per second over the duration of a benchmark run. The server simply counts the size of the input image when the first output image derived from that input has cleared the server (i.e. reached the statistics stage after JPEG encoding). Approximately every five seconds the byte counter was divided by the actual elapsed time as fractions of seconds. The result was the throughput of the server in that interval. For comparison purposes the counter was also divided by the total elapsed time of the benchmark run until that point, also in fractions of seconds. Under different scheduling regimes the five second throughput measurement tended to be bursty, low in one period and then high the next, with variances on the order of 1000 bytes/s. The cumulative throughput (total input bytes / total seconds elapsed) represented a running arithmetic average across bursts. The end cumulative throughput for each benchmark run was the same as the average of the five second throughputs, plus or minus accumulated floating point error and time inaccuracy (at most a few bytes/s).

In addition to the primary metric there are several other metrics of secondary importance, all of which were correlated closely with the primary metric. These secondary metrics include two indicators of client-perceived performance:

- **Client makespan (s)** The client makespan in seconds (i.e. the duration of the benchmark run in seconds) is an indicator of client-perceived performance over an entire benchmark run. The methodology for gathering this metric is described in section 4.2.2 above.
- **Client response rate (responses/s), client response time (s)** Client response rate is simply the client makespan divided by the total number of requests/responses in the benchmark, 4697. The methodology for gathering the client response time in seconds is also described in section 4.2.2. The ninetieth percentile of client response times is preferred to the mean or median as an indicator of how users actually perceive the responsiveness of a system [Wel02].

The secondary metrics for this experiment also included two profiles of server-side behavior:

- **Server CPU use (mean)** The percentage of idle CPU cycles of each core on the server machine was measured at ten second intervals over the entire duration of the benchmark run using `sar`. These percentages were averaged across cores by `sar`, then averaged for a single benchmark run, the latter average excluding startup and shutdown periods (points where the whole machine was more than 90% idle).
- **Server incoming network throughput (max)** During each benchmark run network traffic statistics were sampled at ten second intervals. The interesting metric is the maximum incoming network throughput (in megabits per second) observed during each benchmark run.

4.2.5 Environment

The experiment was executed on the Darwin high performance computing cluster[5] at the University of Cambridge. The experiment required two nodes from the cluster, one node for the server and one for the client. The exact hardware specifications for each node are listed in table 4.3.

Table 4.3: Image processing: environment

Machine type	Dual socket Dell 1950 1U rack mount server
CPU	2 CPUs per node x 2 Intel Woodcrest cores per CPU = 4 cores in total per node @ 3.00 GHz per core
Primary storage	8 GB per node (2 GB per core)
Network	Gigabit Ethernet
Operating system	ClusterVisionOS 2.1 Linux kernel 2.6.9-67.0.4.EL_lustre.1.6.4.3smp x86_64

The job system on Darwin grants exclusive access to nodes, so no other non-idle processes were running on the application or database server nodes during the benchmark runs. In this and subsequent experiments each benchmark run was executed as a separate job. Before the actual benchmark run the client would read the entire image set into memory in order to avoid being bound to the high-latency network file system. After this initial ramp up the client machine was mostly idle.

4.2.6 Results

The results of the experiment are listed in tables 4.4 (the primary metric), 4.5 (client-perceived secondary metrics), and 4.6 (server-side secondary metrics) and figures that show the same data in boxplot form for individual metrics, with each box representing ten runs. The rows of table 4.4 are in descending order of the primary metric, and this order is preserved in the other tables to make comparison simpler. Each of the metrics listed in the tables represents an arithmetic mean of the metric across ten benchmark runs for a given server configuration. In order to compare policies with similar performance in detail the figures for each metric were divided into "top" and "bottom", corresponding to their ranking in table 4.4.

[5] `http://www.hpc.cam.ac.uk/darwin.html`

Table 4.4: Image processing: server throughput in processed bytes/s

Configuration	Mean	Difference in mean	Standard deviation
MG1	12450.92		35.40
Cohort w/o stack	12441.45	-0.08%	42.00
SRPT	12433.82	-0.14%	32.73
Cohort	12240.10	-1.71%	28.05
SEDA w/ Hoard	10843.27	-14%	60.22
SEDA	9490.45	-27%	43.66
SEDA overthreaded	7494.73	-50%	47.43
Thread-per-connection (4)	6120.20	-68%	23.21
SEDA underthreaded	5943.91	-71%	45.82
Thread-per-connection (8)	5845.49	-72%	41.45
Thread-per-connection (16)	4677.09	-91%	47.03

Figure 4.2: Image processing: server throughput (top)

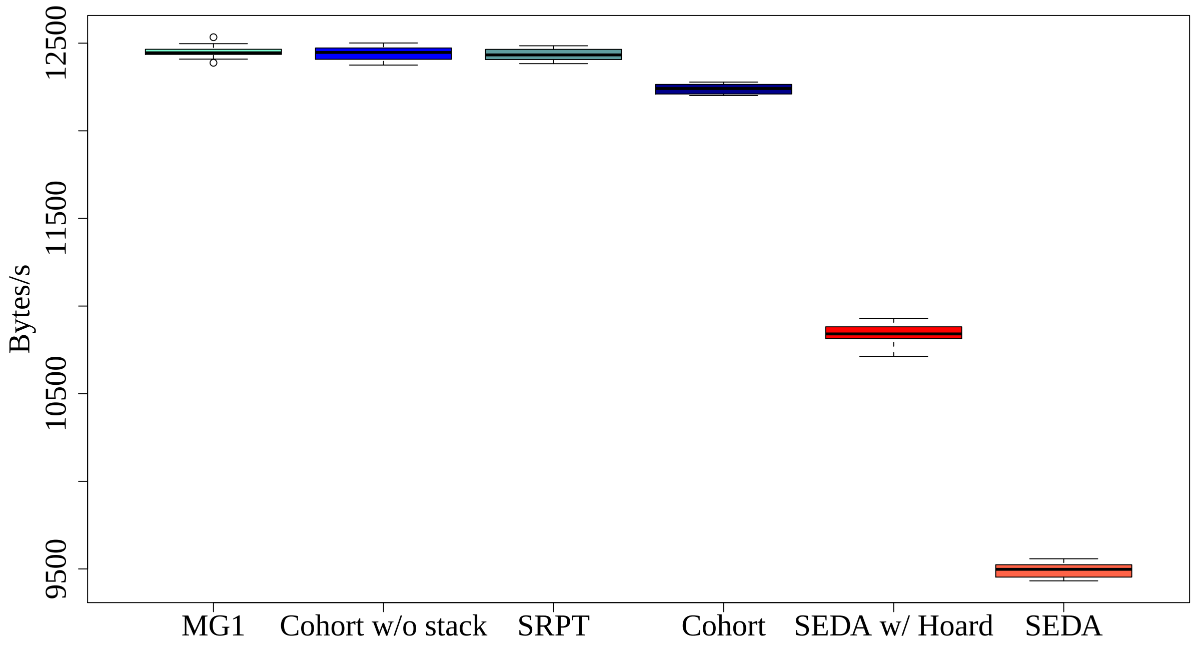


Table 4.5: Image processing: client-perceived performance indicators

Configuration	Makespan (s)	Responses/s	Response time (s) 90th %
MG1	79.14	59.35	0.31
Cohort w/o stack	79.26	59.26	0.32
SRPT	80.76	58.16	0.32
Cohort	80.54	58.32	0.44
SEDA w/ Hoard	91.26	51.47	0.51
SEDA	103.69	43.31	0.53
SEDA overthreaded	131.11	35.83	0.69
Thread-per-connection (4)	161.10	29.16	0.25
SEDA underthreaded	165.77	28.34	0.98
Thread-per-connection (8)	168.33	27.90	0.51
Thread-per-connection (16)	208.75	22.50	1.11

Figure 4.3: Image processing: server throughput (bottom)

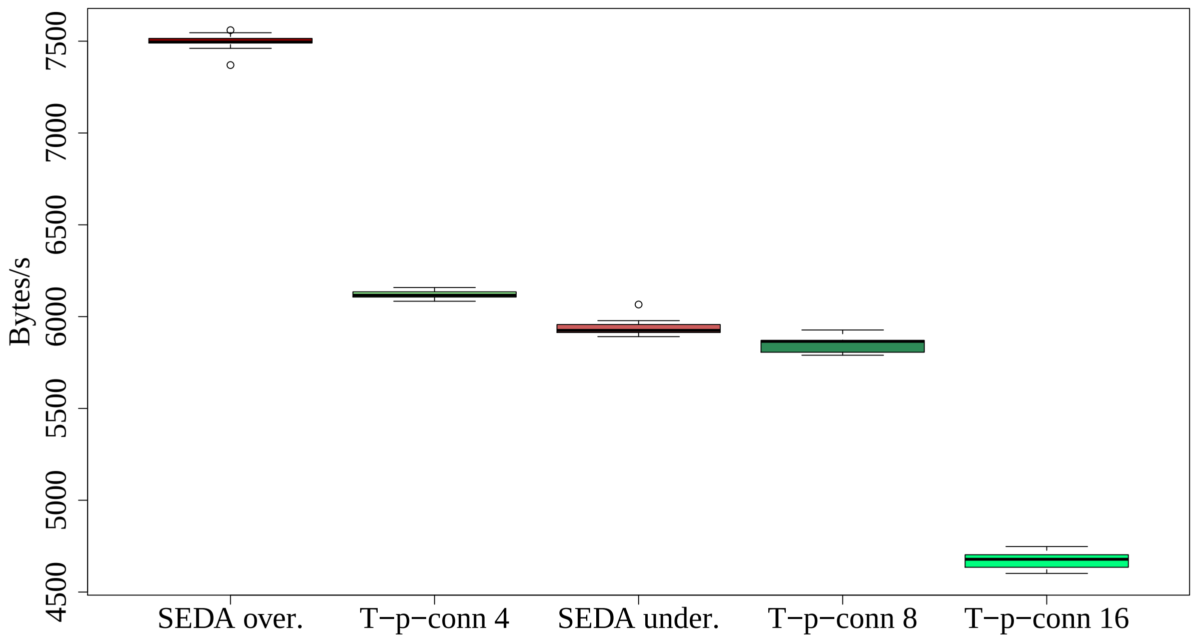


Figure 4.4: Image processing: client makespan (top)

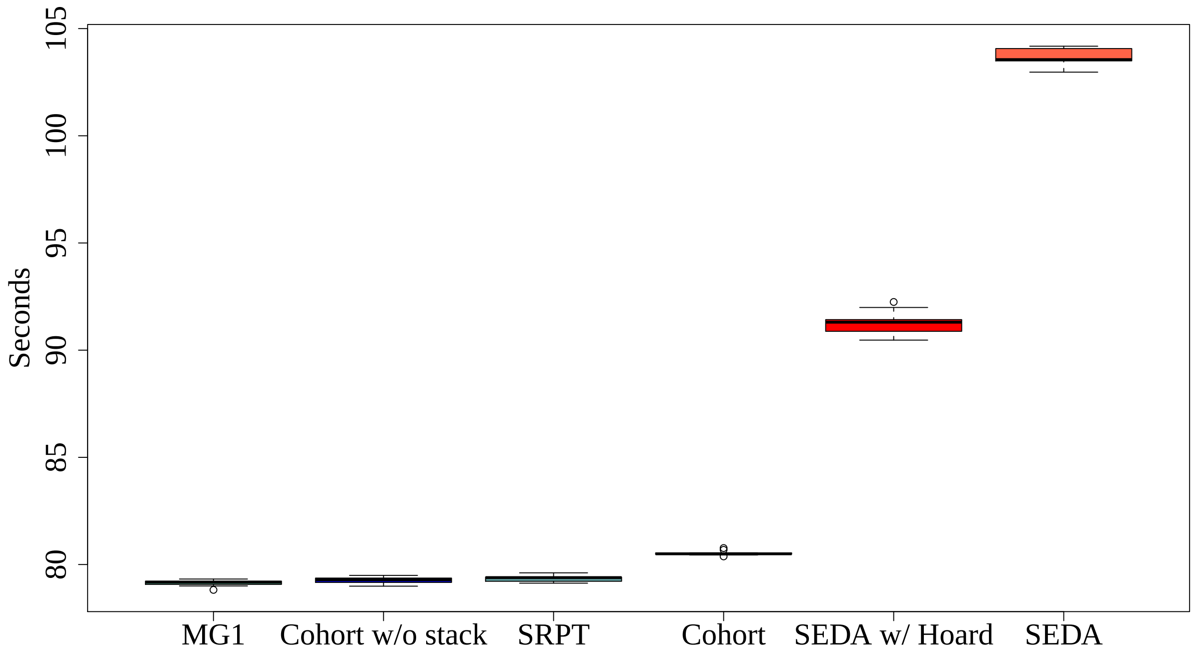


Figure 4.5: Image processing: client makespan (bottom)

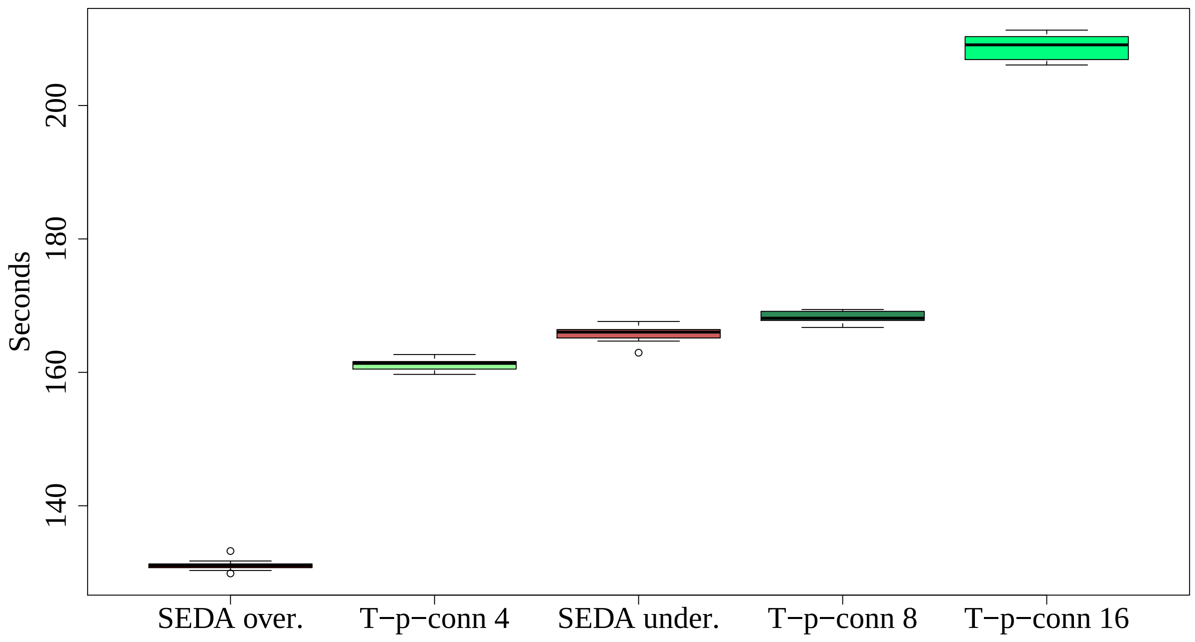


Figure 4.6: Image processing: client-perceived response rate (top)

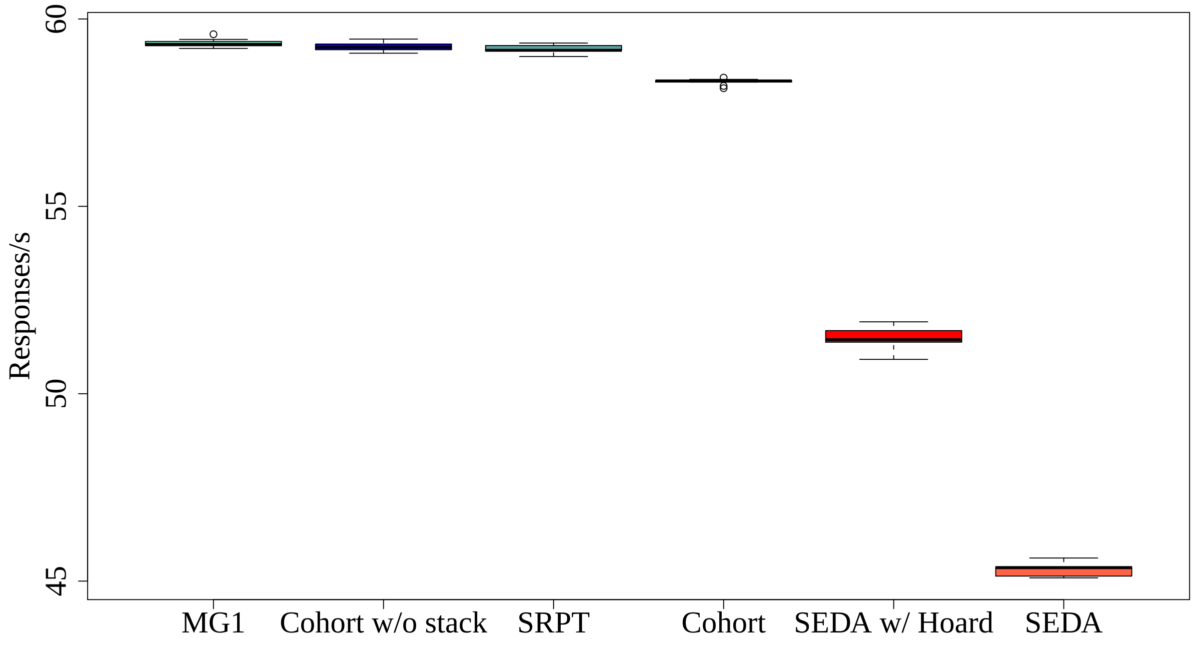


Figure 4.7: Image processing: client-perceived response rate (bottom)

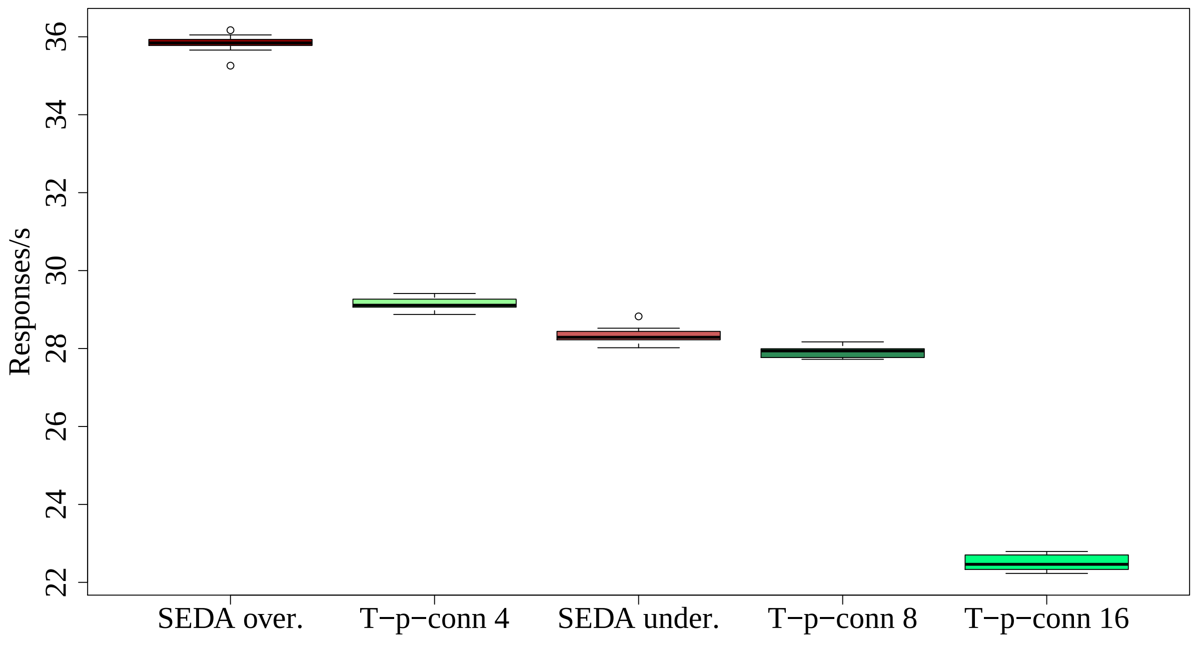


Figure 4.8: Image processing: client-perceived response time (top)

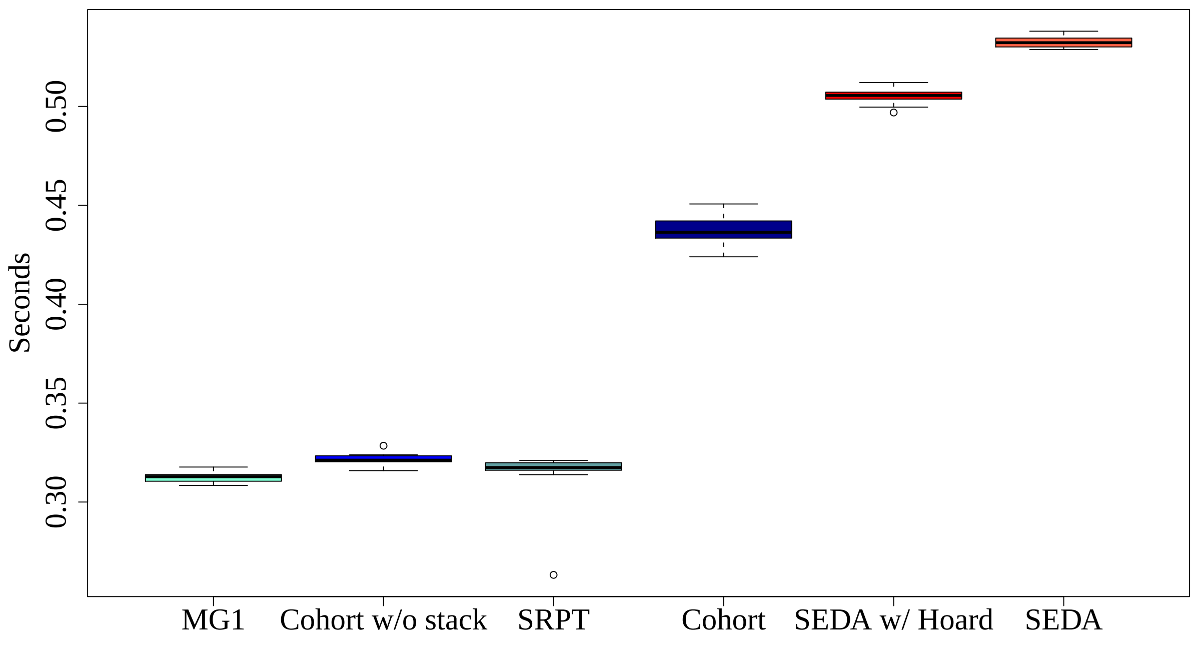


Figure 4.9: Image processing: client-perceived response time (bottom)

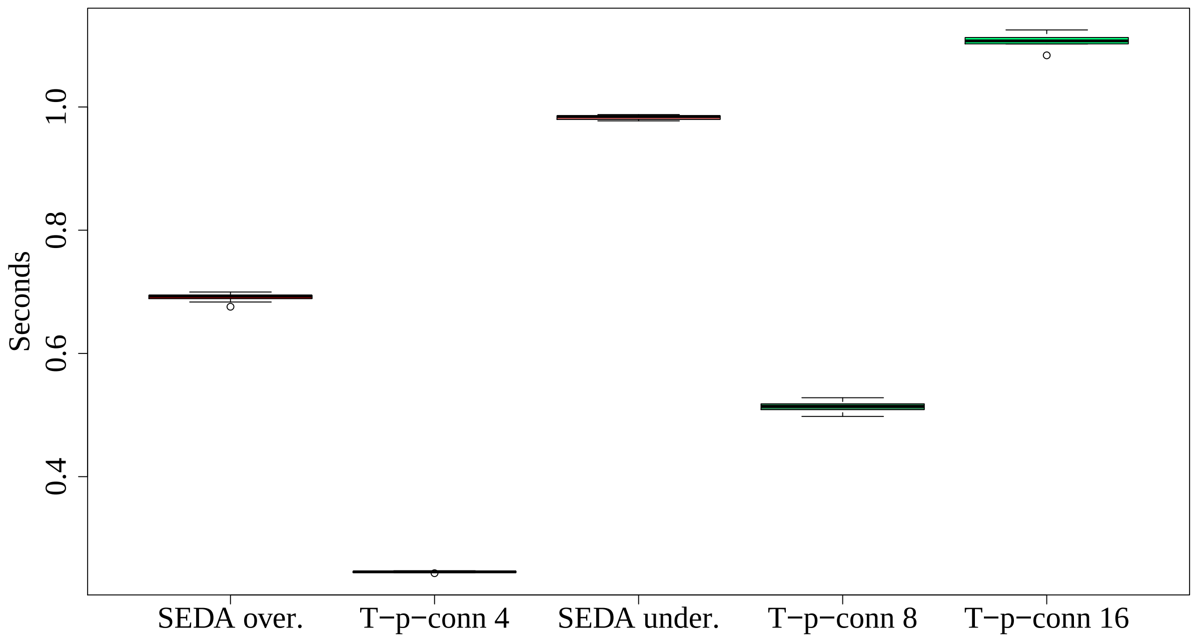


Table 4.6: Image processing: server-side performance indicators

Configuration	Mean CPU use (%)	Max net (Mbps)
MG1	84.65	116.48
Cohort w/o stack	83.36	116.84
SRPT	84.08	117.10
Cohort	86.04	115.82
SEDA w/ Hoard	79.72	101.14
SEDA	73.50	91.08
SEDA overthreaded	81.94	78.45
Thread-per-connection (4)	86.71	60.17
SEDA underthreaded	41.16	61.19
Thread-per-connection (8)	93.57	57.32
Thread-per-connection (16)	96.25	51.79

Figure 4.10: Image processing: server CPU use (top)

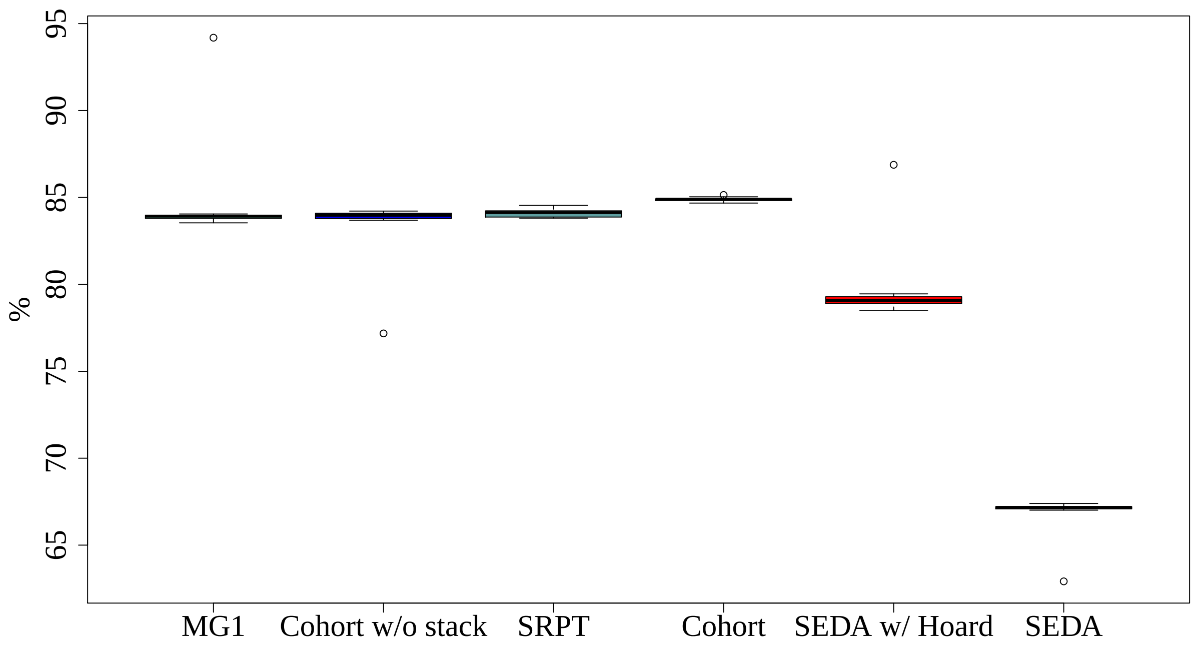


Figure 4.11: Image processing: server CPU use (bottom)

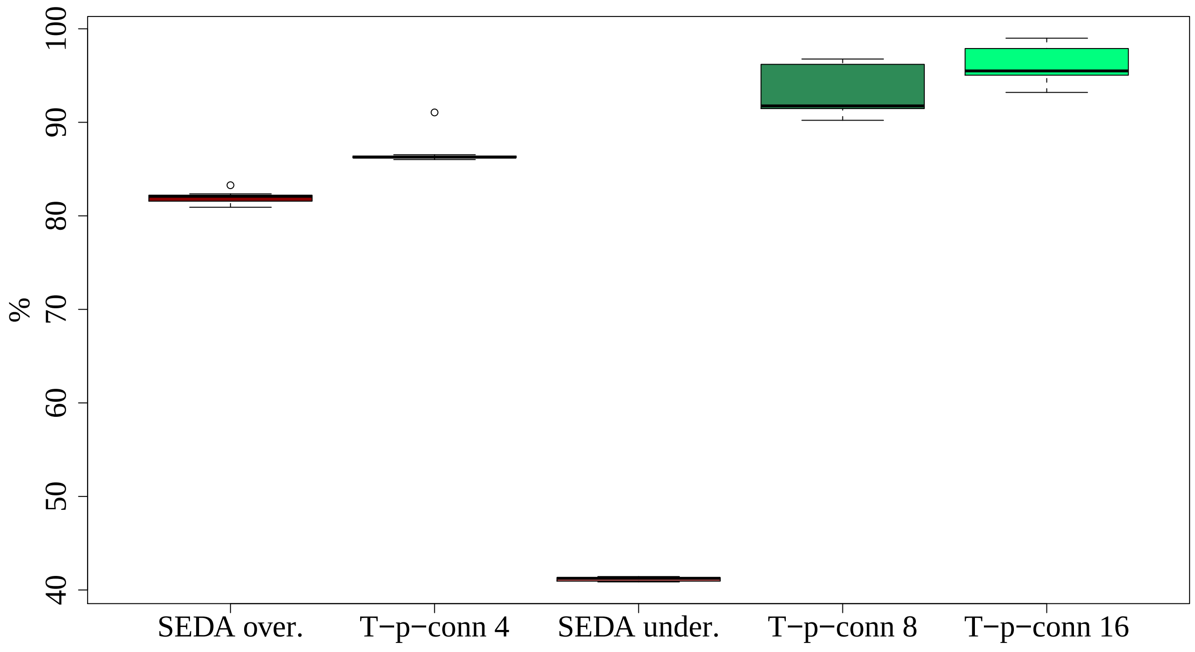


Figure 4.12: Image processing: server incoming network traffic (top)

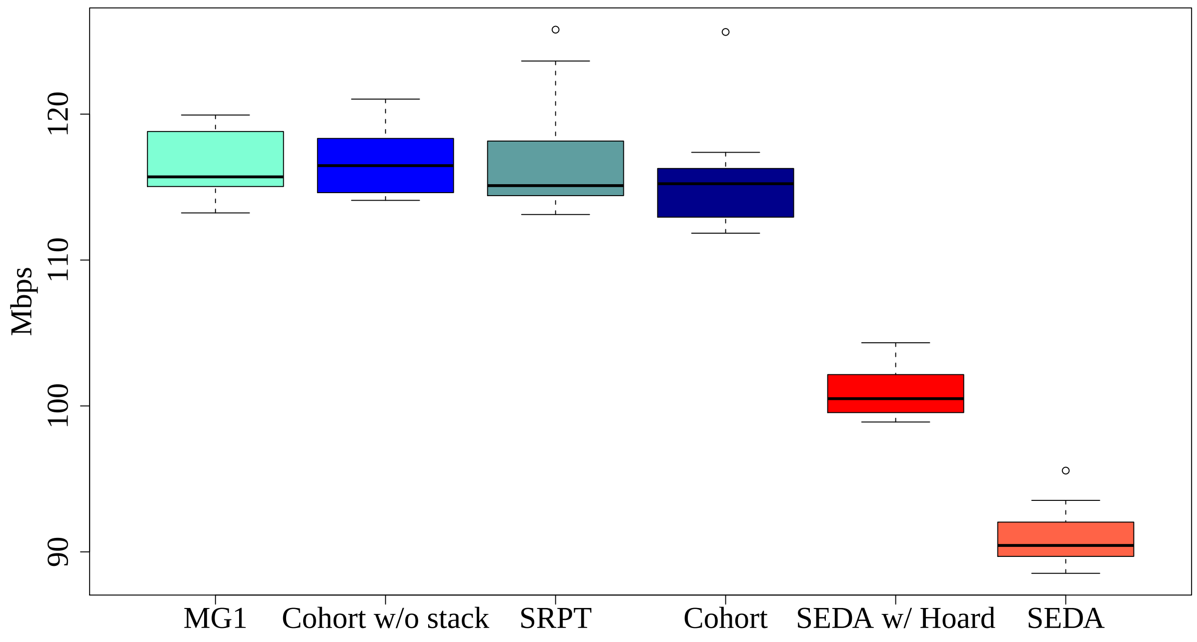
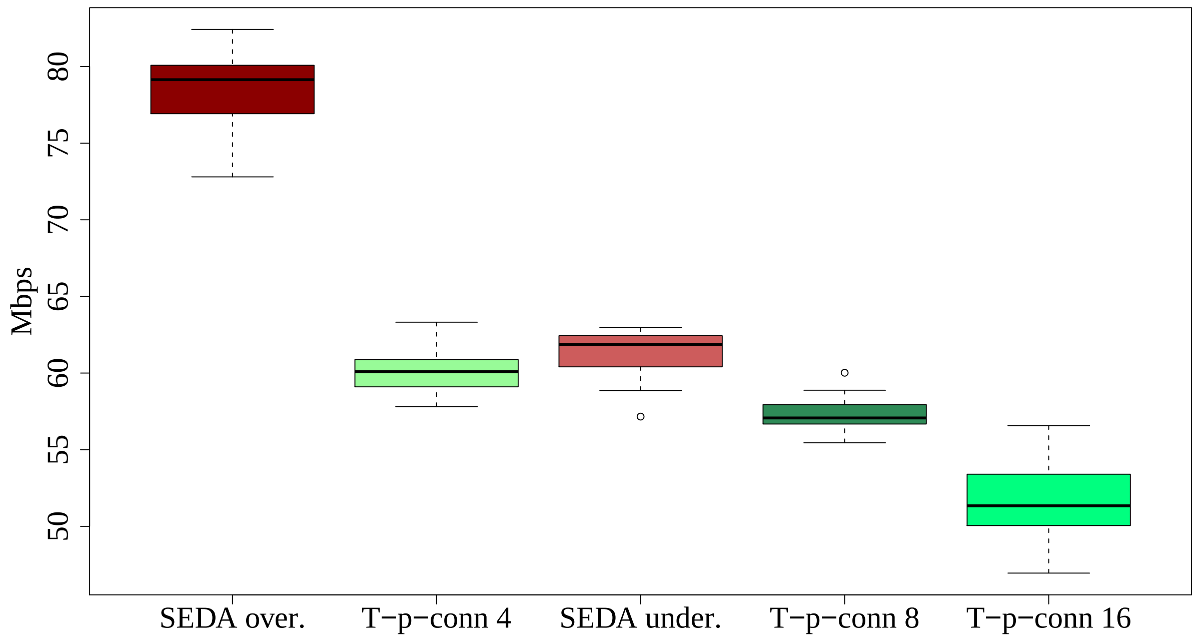


Figure 4.13: Image processing: server incoming network traffic (bottom)



The results of this experiment support my thesis that thread-per-core stage scheduling policies as a class are superior to thread pool-per-stage policies on a throughput-oriented server benchmark. The thread-per-core policies performed 12-14% better according to the primary metric of the experiment, listed in table table 4.4 and figures 4.2 and 4.3. The results also validate my assumption that staged concurrency as a whole performs better on these kinds of benchmarks than conventional kernel thread-per-connection concurrency, a conclusion that is well supported by the literature.

The thread-per-core policies are comparable to one another (within 2%) on the same

Figure 4.14: Image processing: server incoming network traffic, single runs (top)

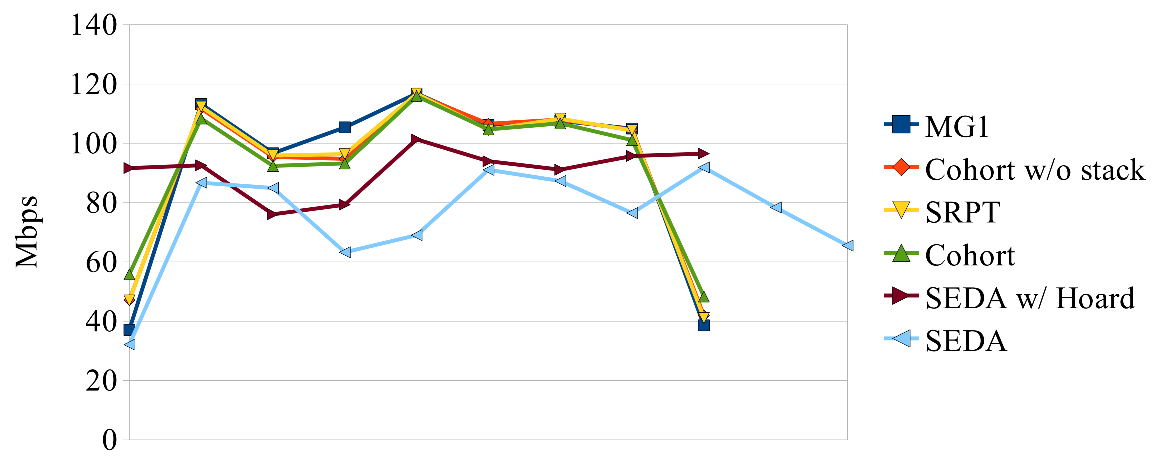
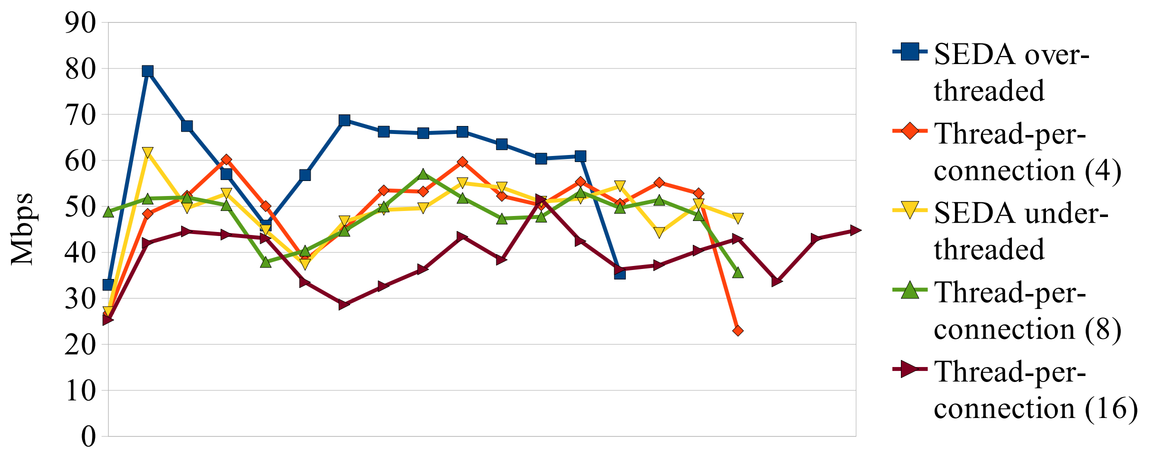


Figure 4.15: Image processing: server incoming network traffic, single runs (bottom)



benchmark, despite my expectation that the MG1 heuristic of visiting heavily-loaded stages more frequently should result in better performance than simply iterating over the array of stages in a fixed order. The Cohort scheduling/wavefront algorithm performs slightly better without per-core stacks than with them. Assuming this is not an artifact of the implementation, it could be attributed to the delays incurred by this system: the events on a per-core stack can only be handled by that core, which means that they must wait in the stack until the core has polled the other stages in the system.

Increasing or decreasing the size of the thread pools in SEDA thread pool-per-stage concurrency has an obvious impact on the throughput of the server. When there are many more threads than stages, the contention between threads and the fairness policy of the kernel scheduler drags down the total throughput of the server and increases CPU use (figure 4.11). Conversely, when there are too few threads they cannot fully exploit the available processing power (figure 4.11), even if the number of threads is the same as the number of cores: the CPU requirements of the different stages are not equal, and simply assigning one thread/core to each stage is far from optimal. In other experiments (see chapter 5) I have observed situations where a thread-safe stage could bear more load than a single thread/core could handle, but adding another thread was overkill and upset the balance of the system by diluting the CPU shares of other stages.

As expected, the client-perceived performance indicators listed in table 4.5 are closely correlated with the primary metric. Client makespan (figures 4.4 and 4.5) was initially used in test runs to predict the performance of the various server configurations. Client response rates (figures 4.6 and 4.7) also have a linear relationship with server throughput, since requests were not reordered in the server.

Client-perceived response times (figures 4.8 and 4.9), though not the primary metric, are also interesting in that they reflect the strengths and weaknesses of the tested policies. The kernel thread-per-connection server with four threads, one for each core on the machine, exhibited the lowest response times of any of the server configurations (figure 4.9). With more threads in the system there was more contention between threads, which increased CPU use (figure 4.11) while decreasing throughput (figure 4.3). The SRPT thread-per-core policy was tied for the second lowest response times (figure 4.2), in keeping with its preference for clearing stages at the end of the stage graph.

Table 4.6 lists the two server-side performance indicators, CPU use and incoming network traffic. The CPU use of the different scheduling policies was comparable (figures 4.10 and 4.11), except where an algorithm was handicapped by contention (kernel thread-per-connection with more than 4 threads, SEDA overthreaded, SEDA without the Hoard memory allocator) or too few threads (SEDA underthreaded).

As expected, measurements of incoming network traffic tracked server throughput at a coarser grain than that observed by the client as the primary metric. The maximum incoming network throughput of the servers is plotted in figures 4.12 and 4.13. Figures 4.14 and 4.15 show observed incoming network traffic at the various servers over the duration of a run (excluding startup and shutdown) for single, representative runs. Note that the servers with lower throughput/network traffic took longer to complete the benchmark and thus have more points in the two figures.

4.3 Experiment: comparing stage scheduling policies in a site search engine

The mass of content available on the Web and the absence of a unified structure can make finding specific content difficult. In the last decade users have come to rely heavily on search engines to filter Web content and bypass site-specific navigation. Although general, web-wide search engines have proved quite adept at this task, the largest and most complex web sites often include site-specific search engines as well. These are designed and optimized to exploit the structure and nature of content on the site and tuned to serve specific kinds of user queries. For example, the site search engine for a hardware manufacturer may attempt to match variations of a product name or model, variations a general keyword-based search engines would likely miss because they are not listed on any page.

The second experiment in this chapter involves a page lookup / search engine that is optimized for a specific site, the English version of Wikipedia[6]. The purpose of this experiment is to compare the stage scheduling policies described in chapters 2 and 3,

[6] `http://en.wikipedia.org/`

including thread pool-per-stage policies (a.k.a. SEDA) and thread-per-core policies (Cohort scheduling, Color, MG1, SRPT, DBR). Thread-per-connection concurrency was not evaluated in this experiment.

4.3.1 Workload

The ideal benchmark for this experiment would consist of a data set comprising the entire content of a real web site (such as Wikipedia) and a trace of real queries seen by the site's existing search engine, replayed against the server. However, in order to test the efficiency of the CPU scheduler the server must be CPU-intensive, which generally implies that the server must be able to work almost entirely from main memory. The working set of this server consists of various indices on the content of the site, and several of these indices grow linearly with the size of the content base. Due to this working set constraint as well as the unavailability of actual query data, the benchmark falls somewhat short of perfect realism:

- In order for the working set of indices to fit in memory a two gigabyte file of circa 1.5 million page abstracts was substituted for the 60+ gigabyte full text of the English Wikipedia[7]. Each abstract includes a page title, a short summary of the page, and the page's section headers. This information was sufficient to match most keyword queries with realistic results.
- The query set was synthesized from the logs of page hit counts. The hit counts for each page are aggregated per hour and archived per day[8]. The majority of page requests were referrals from popular web-wide search engines, and were thus for exact page titles, a fact which was exploited by the server. However, the logs also included requests for pages that do not exist, variations on page titles, and apparent conjunctive searches, which provided ample fodder for the server.

Both tradeoffs imply that the server is not a conventional site search engine that indexes the site's full content and answers real queries on that content. Rather, the server more closely resembles the front end of a Content Management System. The front end's main function is to match request URIs against a database of pages, but the server will also suggest "related pages" in the absence of an exact match, rather than simply returning a missing page response as a normal web server would.

4.3.2 Client software

Thirty-two instances of `httperf` [MJ98] on eight machines acted as clients in this experiment. The clients issued requests in a closed loop [SWHB06] from a pre-generated file of request URIs, with each client utilizing several hundred connections to the server. As in the previous experiment, the goal was to push the server to peak performance and observe its throughput, rather than subjecting the server to overload in an open loop.

[7]Both the full text and abstracts file are available at `http://en.wikipedia.org/wiki/Wikipedia:Database_download`. The July 24th, 2008 abstracts file was used for this benchmark.

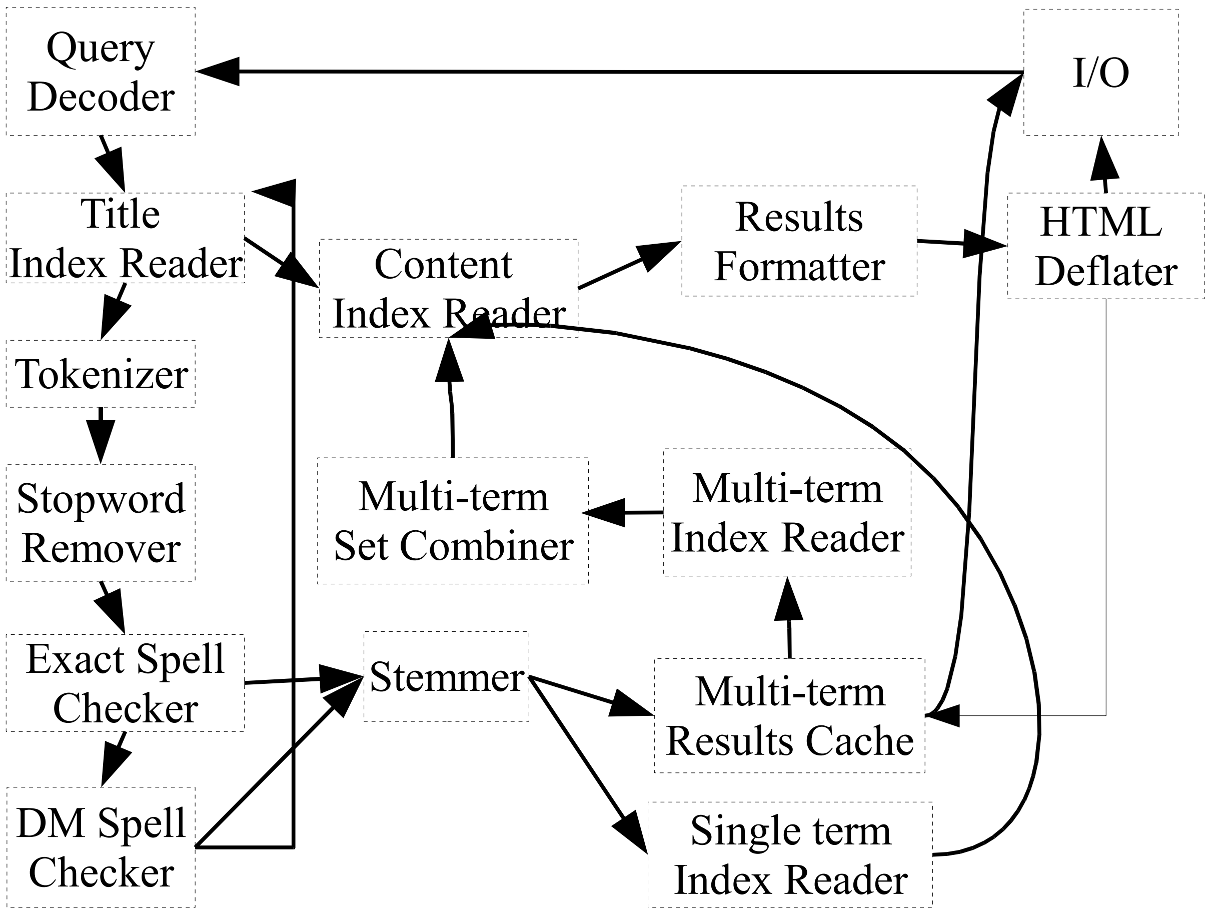
[8]These logs are available at `http://dammit.lt/wikistats/`. The query set for this benchmark was synthesized from the logs for August 1, 2008.

A benchmark run consisted of the 32 clients concurrently iterating through unique (per-client) lists of 10,000 requests to warm up the server (i.e. retrieve the indices from memory) then switching to a second unique list of 10,000 requests for the observed run. The 64 different lists of 10,000 requests apiece were generated by calculating the top 100,000 pages according to the day's page count statistics (excluding special pages and files that are not present in the abstracts file, such as `Special:Random`), putting these pages in a range (`page id * the number of hits per day`), and drawing 80,000 random numbers against the range. The page request distribution is heavy-tailed, with the top request URIs all variations of page titles (e.g. `Main_Page`, the most-requested page).

4.3.3 Server software

Figure 4.16 shows the stage graph of the benchmark server. The graph has 14 stages in three major paths, which correspond to three types of matches: exact title matches, single term searches, and multi-term searches. Error paths (such as e.g. searches with no non-stopwords) are not shown.

Figure 4.16: Site search engine



In the normal case request processing logic proceeds through the server as follows:

1. **I/O**: accepts a connection and reads HTTP requests

2. **Query decoder**: checks the request URI for the right prefix (`/wiki/`), URI-decodes it, and (for ASCII requests) transforms the URI a lower case copy, and sends the newly-minted query to the title index reader
3. **Title index reader**: compares the query as decoded against a index of page titles, both lower and normal case; exact matches are sent to the content index reader, all other queries are sent to the tokenizer
4. **Tokenizer**: decomposes the query into tokens using a lexer from the Robust Accurate Statistical Parsing (RASP) library [BCW06]
5. **Stopword remover**: removes common English words ("the", "and") from the list of query tokens and sends this list to the exact spell checker
6. **Exact spell checker**: compares the list of non-stopword query terms to an index of terms present in the abstracts file; when all words are spelled correctly (the normal case) the query is sent to the stemmer, otherwise it is sent to the DM spell checker
7. **DM spell checker**: uses the Double Metaphone algorithm [Phi00] to replace misspelled terms with similar-sounding words found in the abstracts library; the query is then returned to the title index reader for rechecking, but bypasses spell checking on a subsequent title index miss
8. **Stemmer**: reduces the query terms to their stems (e.g. removing "ing" at the end of verbs) using a variant of the Porter stemming algorithm [Por80], then forwards the query to either the single term index reader or the multi-term results cache
9. **Single term index reader**: checks the term index for the single query term and returns the first ten matching document IDs, which are then sent to the content index reader
10. **Multi-term results cache**: hashes the stemmed query terms and compares the hash to a small (< 64M) cache of previously-generated multi-term query results (e.g. deflated HTML); cache hits are responded to immediately, while cache misses go on to the multi-term index reader
11. **Multi-term index reader**: iterates through the array of query terms, compiling a list of document IDs that match any term in the abstract
12. **Multi-term set combiner**: calculates the intersection of the sets of document IDs associated with each term and sends the first 10 document IDs in this combined set to the content index reader
13. **Content index reader**: looks up document IDs from the title or term index reader in an index of page titles and abstracts and produces a list of matching title/abstracts
14. **Results formatter**: uses the `ctemplate` library to format the list of page titles and abstracts as an HTML table and forwards the resulting HTML to be deflated
15. **HTML deflater**: compresses the HTML response body using zlib[9] and responds to the HTTP request

[9]`http://www.zlib.net/`

The title, term, and spell checking indices are Berkeley DB[10] B-trees pre-built by a separate staged program, which re-uses many of the stages listed above. The term index is a simple inverted index, using duplicate keys (one key per term) in lieu of more advanced schemes such as e.g. compressing and packing values in order to minimize database page overflow[11]. The algorithm for combining document IDs is also a simple mix of sorting and duplicate counting; the problem of efficient integer set intersection as it arises in search engines is well known and has been addressed in various clever ways, see e.g. [ST07]. Like any benchmark implementation, the server is a compromise between realism and ease of development.

As in the image processing server, not all of the potential stages in figure 4.16 were useful in practice, and having more stages proved a hindrance to some of the scheduling policies in cases where an event handler did not do enough work per event (in terms of CPU cycles) to justify having a separate stage.

For this experiment the thread pool-per-stage policies were evaluated with three different configurations: “normal”, in which the most CPU-intensive stage, the multi-term set combiner, was given one thread per core and all other stages were given a single thread each; “overthreaded”, in which every potential stage in the figure was a stage and each thread-safe stage was given one thread per physical core; and “underthreaded”, in which less CPU-intensive stages such as the stemmer and spell checkers were not made into stages (i.e. they were pass-through), the multi-term set combiner was given one thread per core, and the remaining stages were each given a single thread.

The thread-per-core policies were evaluated in only two configurations: “normal”, in which all stages were visited by every core; and ”underthreaded”, in which the less CPU-intensive stages were not made into stages while all other stages were visited by every core.

4.3.4 Metrics

The primary metric in this experiment is the response rate observed by the client. Response rates are an appropriate metric here because the server always returns responses of approximately the same size (less than one kilobyte), which makes the response rates under different scheduling regimes comparable.

The burstiness of some scheduling policies is reflected in differences between mean response rates and the maximum response rate across measurement periods. `httperf` samples response rates in five second intervals. The final mean response rate displayed by `httperf` is the mean of the response rates in every sample, while the maximum is the maximum response rate seen in any sample. Both metrics are included in the results below, in order to indicate differences in mean and maximum throughput. As in the image processing server experiment, all of the results are averages over ten runs.

In the majority of server configurations CPU use stayed around 80% on average during a benchmark run. The exceptions were overthreaded SEDA configuration, which utilized 90-100% CPU due to increased contention. The amount of network I/O was proportional to the number of responses and is not shown in the results.

[10]`http://www.oracle.com/technology/products/berkeley-db/index.html`

[11]See [MRYGM01] for an analysis of these schemes and other challenges in implementing search engines using off-the-shelf software

In order to better understand the performance differences between stage scheduling policies the total numbers of L1 data cache and L2 instruction cache misses across all cores were measured during each benchmark run. The total number of L2 data cache misses was not available on the server machine.

4.3.5 Environment

This experiment was run on the Alibaba cluster of Linux machines at the Konrad Zuse Institut in Berlin[12]. The specifications of a single client machine are listed in table 4.7. The experiment required eight identical client worker nodes, each hosting four instances of `httperf`.

Table 4.7: Site search engine benchmark environment: clients

Machine type	Dell PowerEdge (tm) 1950 Xeon E5420
CPU	2 processors per machine x 2 Intel Woodcrest cores per processor = 4 cores per machine @ 2.5 Ghz per core, FSB 1333 MHz
Primary storage	8 GB
Network	Doubled Gigabit Ethernet
Operating system	SUSE Linux Enterprise Server 10 Linux kernel 2.6.16.53-0.16-smp x86_64

The server was a Sun Solaris/amd64 machine connected to the cluster. The specifications of this machine are listed in table 4.8.

Table 4.8: Site search engine benchmark environment: server

Machine type	Sun Blade X8440
CPU	2 processors per machine x AMD Opteron quad-core (835x Series) processors = 8 cores @ 2.3 Ghz per core
Primary storage	32 GB
Network	Gigabit Ethernet
Operating system	SunOS 5.10

4.3.6 Results

The results of the experiment are shown in tables 4.9, 4.10, 4.11, 4.12, and 4.13 and boxplot figures covering the same data, with each figure separated into top and bottom performers according to their rank in table 4.9. Each box represents the ten runs for a given configuration.

[12]`http://www.zib.de/cluster-user/view`

Table 4.9: Site search engine: mean response rate (responses/s)

Policy	Configuration	Mean	Difference in mean	Standard deviation
Color	normal	922.31		0.86
Color	underthreaded	921.98	-0.04%	1.66
SEDA	normal	912.89	-1.03%	1.37
SEDA	underthreaded	912.48	-1.07%	1.13
SRPT	underthreaded	883.80	-4.26%	3.53
SRPT	normal	883.48	-4.30%	1.43
MG1	normal	871.49	-5.67%	2.88
MG1	underthreaded	871.21	-5.70%	2.08
Cohort	normal	846.46	-8.58%	12.49
Cohort	underthreaded	843.39	-8.94%	8.55
DBR	normal	800.82	-14.10%	41.24
DBR	underthreaded	795.77	-14.73%	19.41
SEDA	overthreaded	661.13	-32.99%	26.64

Figure 4.17: Site search engine: mean response rate (top)

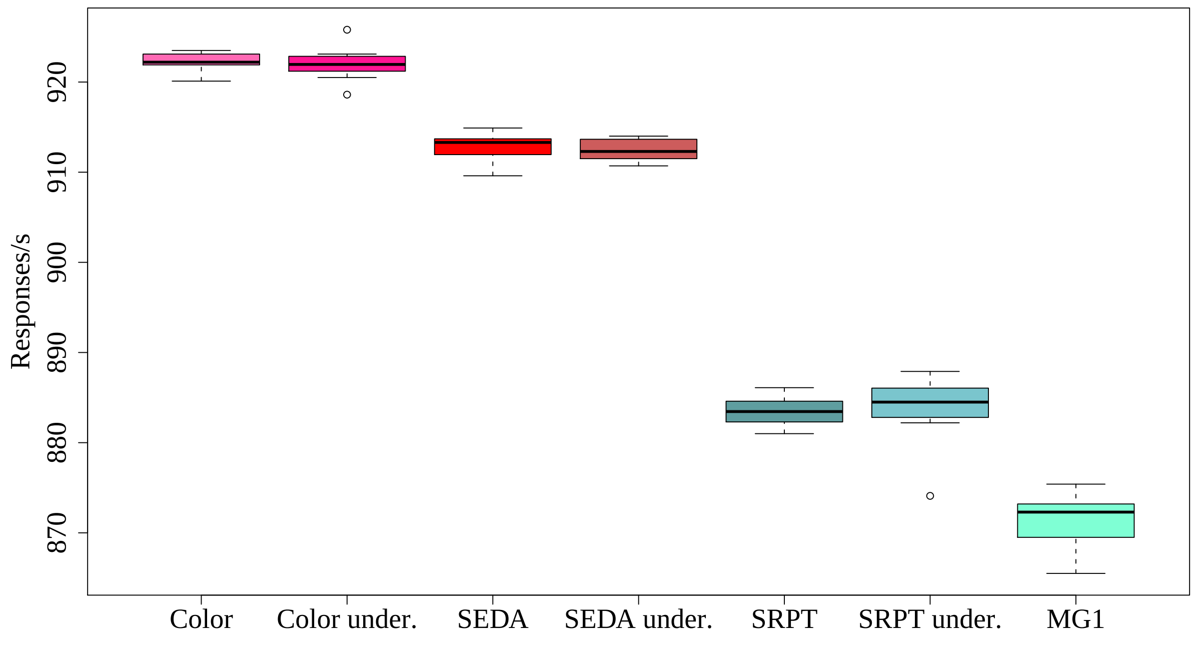


Figure 4.18: Site search engine: mean response rate (bottom)

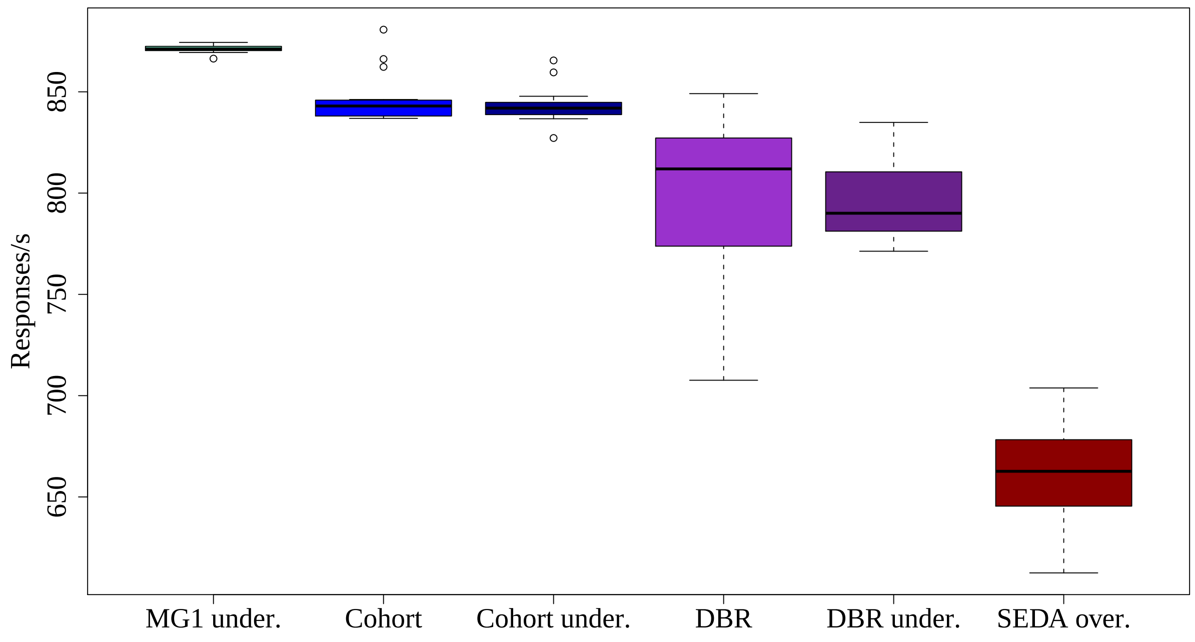


Table 4.10: Site search engine: max response rate (responses/s)

Policy	Configuration	Mean	Difference in mean	Standard deviation
Color	underthreaded	1556.37		13.74
Color	normal	1553.57	-0.18%	13.30
SEDA	normal	1490.33	-4.34%	14.43
SEDA	underthreaded	1463.60	-6.14%	29.29
SRPT	normal	1406.98	-10.08%	13.12
SRPT	underthreaded	1405.73	-10.17%	21.11
MG1	normal	1395.31	-10.91%	10.53
MG1	underthreaded	1387.22	-11.49%	16.35
Cohort	normal	1381.92	-11.87%	20.17
Cohort	underthreaded	1380.15	-12.00%	17.98
DBR	normal	1308.14	-17.33%	63.22
DBR	underthreaded	1301.52	-17.83%	79.65
SEDA	overthreaded	1017.53	-41.87%	63.15

Figure 4.19: Site search engine: max response rate (top)

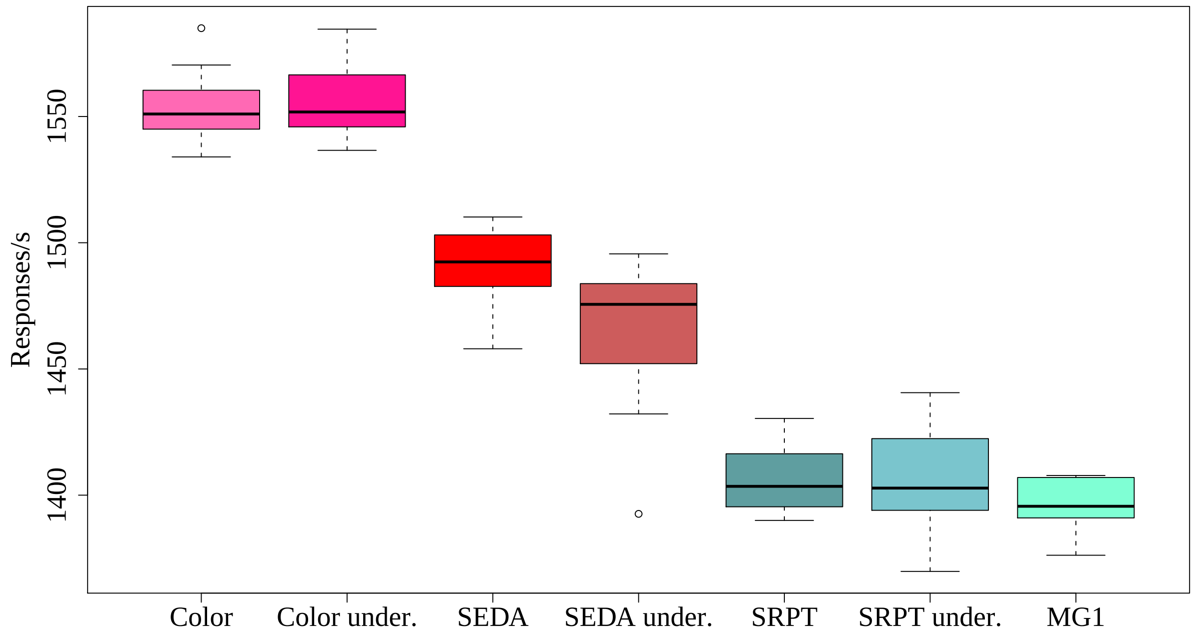


Figure 4.20: Site search engine: max response rate (bottom)

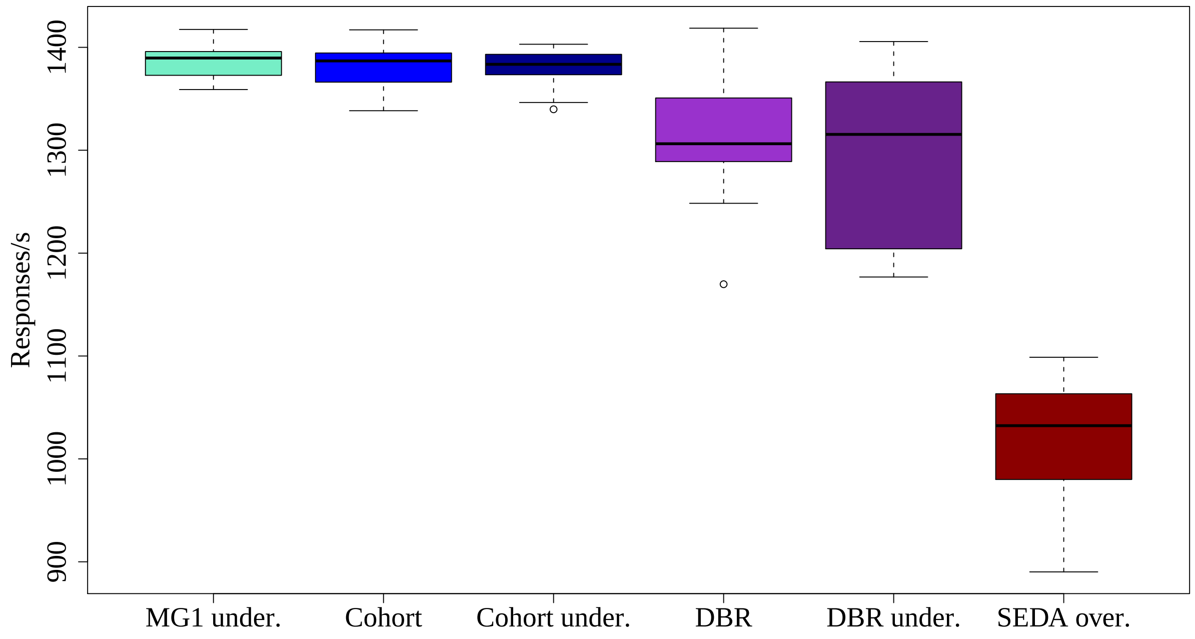


Table 4.11: Site search engine: response time (ms)

Policy	Configuration	Mean	Difference in mean	Standard deviation
Color	underthreaded	32.36		2.90
Color	normal	33.97	+4.85%	1.22
SEDA	underthreaded	199.67	+144.21%	20.92
SEDA	normal	200.08	+144.31%	17.89
SRPT	underthreaded	719.63	+182.79%	36.10
SRPT	normal	739.81	+183.24%	22.40
MG1	underthreaded	941.01	+186.70%	35.52
MG1	normal	943.22	+186.73%	48.08
Cohort	normal	1351.58	+190.65%	251.16
Cohort	underthreaded	1418.06	+191.08%	154.87
DBR	normal	2112.54	+193.97%	469.46
DBR	underthreaded	2377.18	+194.63%	373.04
SEDA	overthreaded	3310.98	+196.13%	1032.57

Figure 4.21: Site search engine: response time (top)

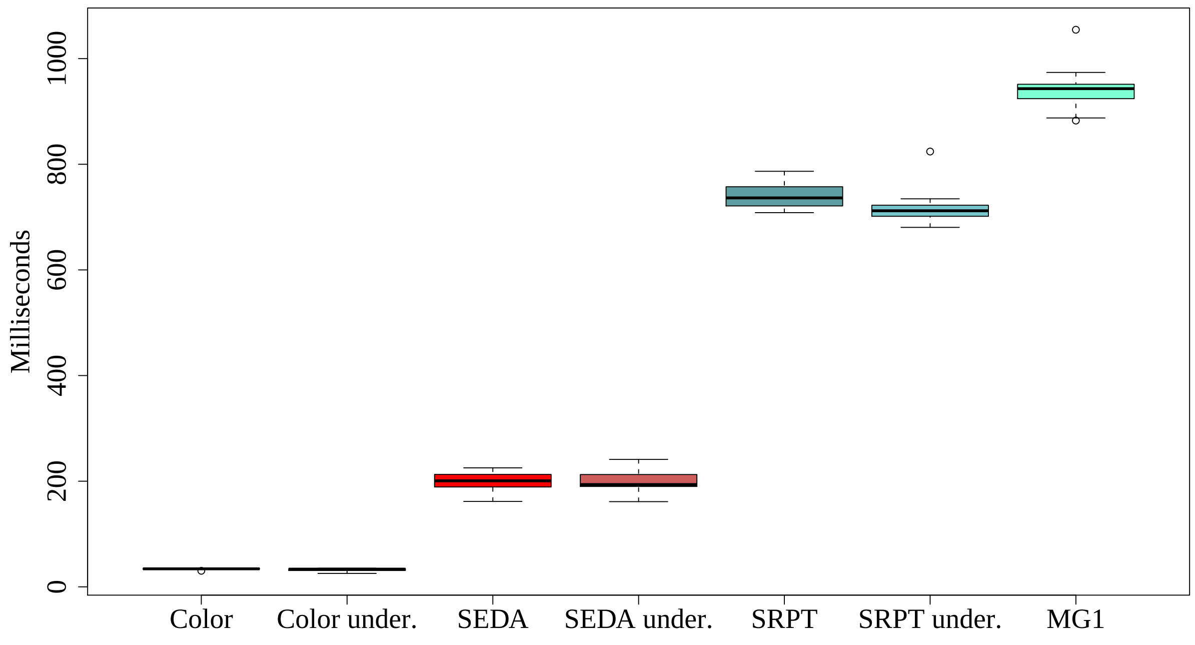


Figure 4.22: Site search engine: response time (bottom)

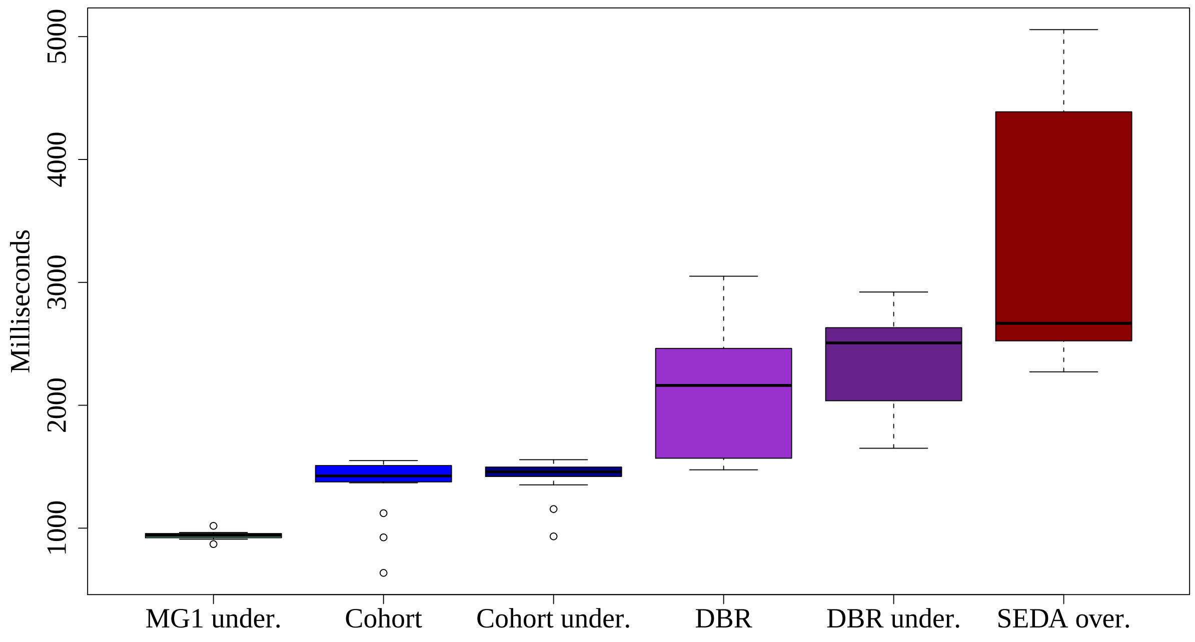


Table 4.12: Site search engine: L1 data cache misses

Policy	Configuration	Mean	Difference in mean	Standard deviation
MG1	underthreaded	3855586.27		241220.45
Cohort	normal	3959960.00	+2.67%	394522.74
Cohort	underthreaded	4041776.06	+4.72%	281313.69
Color	normal	4060666.00	+5.18%	303924.71
Color	underthreaded	4184585.75	+8.18%	400842.21
MG1	normal	4363840.44	+12.37%	1438273.43
SRPT	underthreaded	4584919.09	+17.28%	234690.90
SRPT	normal	4813122.33	+22.09%	785334.52
SEDA	normal	5086518.40	+27.53%	494204.05
SEDA	overthreaded	5649068.13	+37.74%	797723.57
SEDA	underthreaded	5868888.73	+41.41%	842568.92
DBR	normal	6113364.00	+45.30%	2925928.88
DBR	underthreaded	11081368.00	+96.75%	7825220.05

Figure 4.23: Site search engine: L1 data cache misses (top)

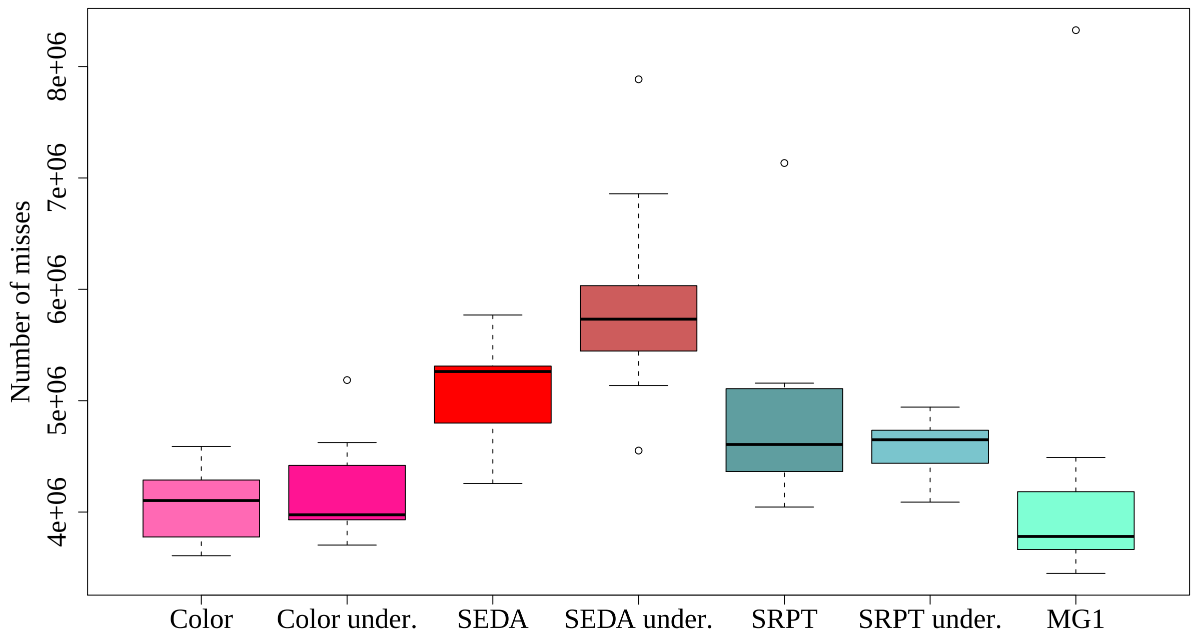


Figure 4.24: Site search engine: L1 data cache misses (bottom)

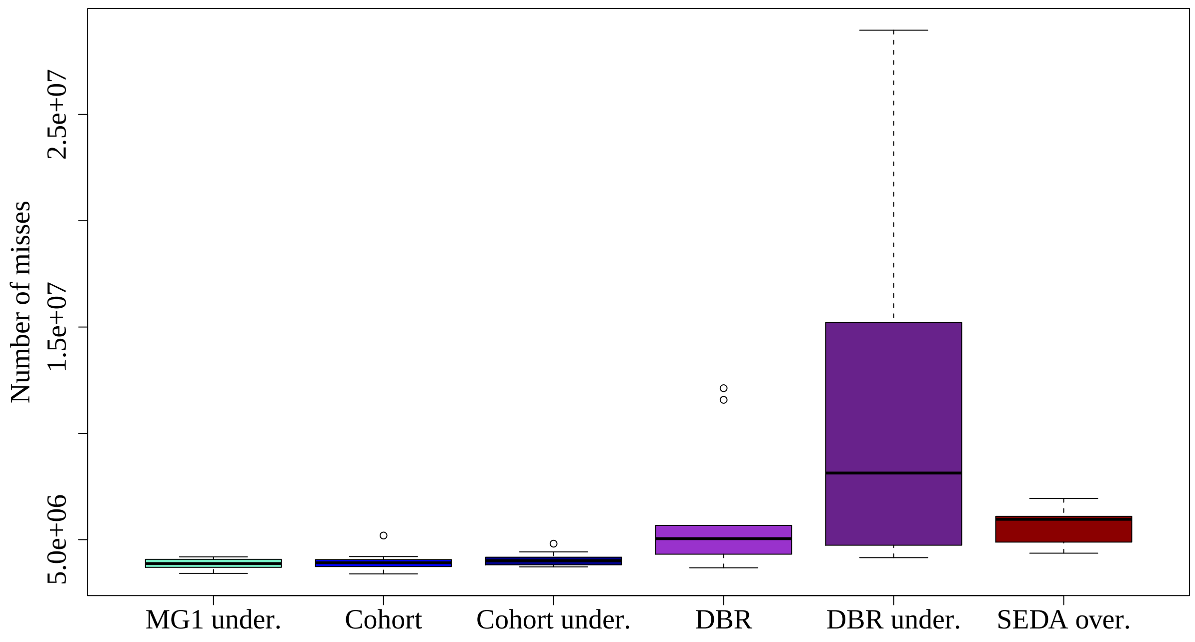
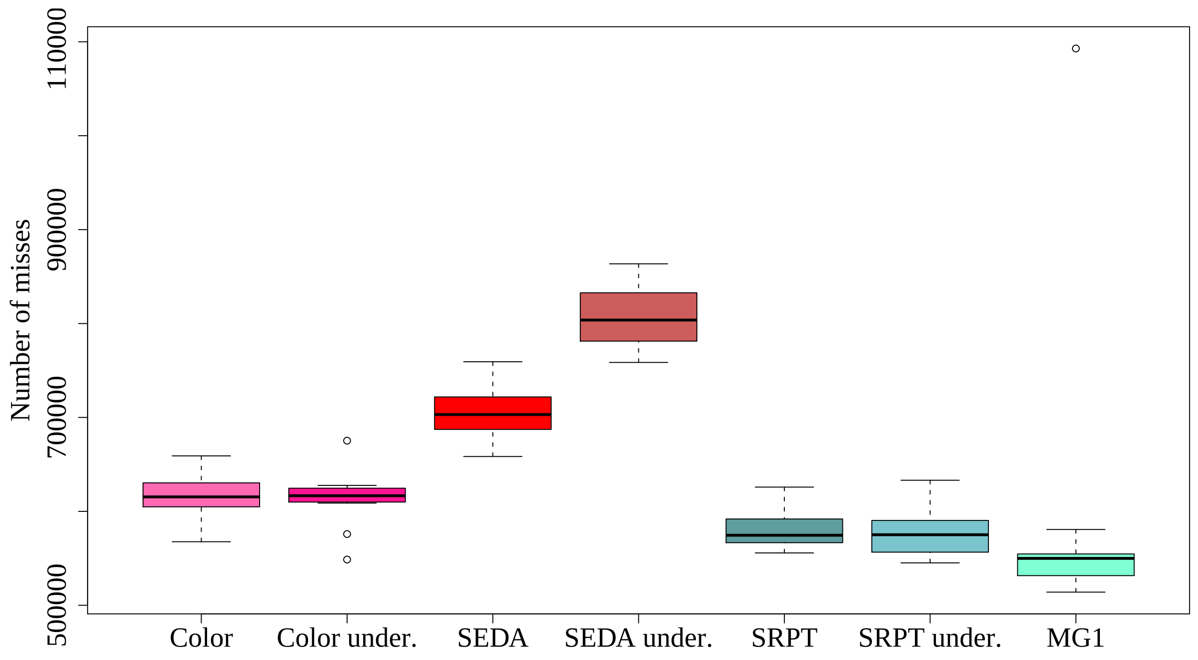


Table 4.13: Site search engine: L2 instruction cache misses

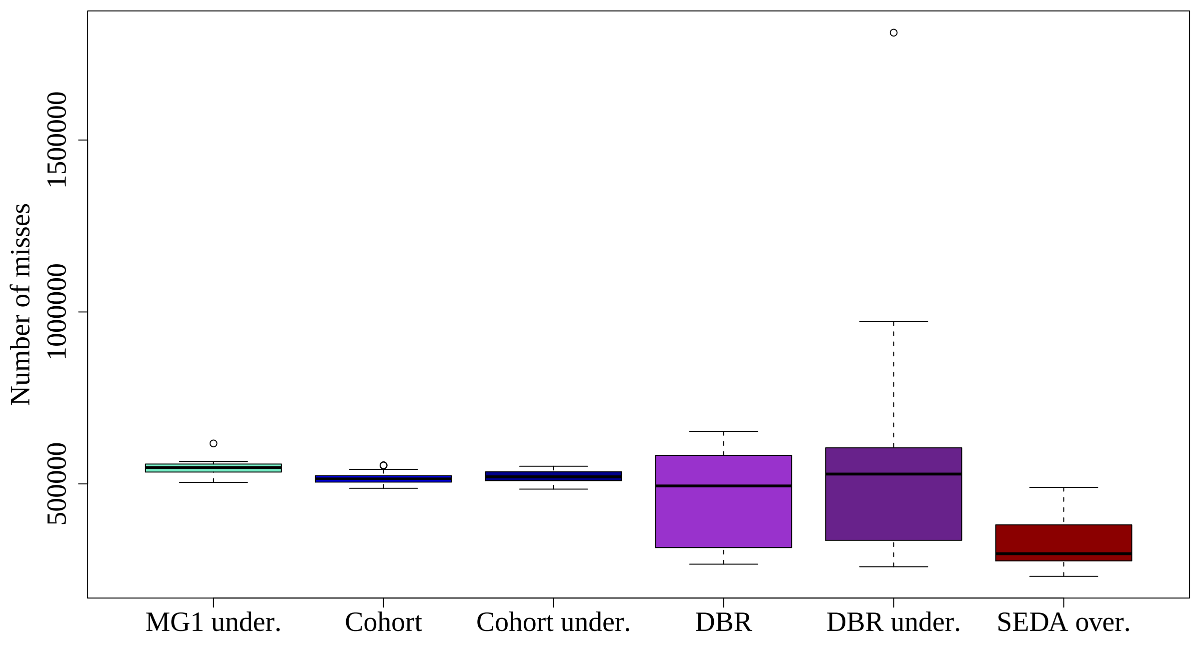
Policy	Configuration	Mean	Difference in mean	Standard deviation
SEDA	overthreaded	328659.13		80170.52
DBR	normal	464740.90	+34.30%	144208.02
Cohort	normal	517747.13	+44.68%	18749.74
Cohort	underthreaded	520804.75	+45.24%	17437.02
MG1	underthreaded	549000.91	+50.21%	26979.93
SRPT	underthreaded	575649.55	+54.63%	24975.58
SRPT	normal	579839.50	+55.30%	19301.56
MG1	normal	604776.44	+59.16%	173539.03
Color	underthreaded	613794.58	+60.51%	28990.39
Color	normal	616964.15	+60.98%	22263.77
DBR	underthreaded	641884.30	+64.55%	435059.30
SEDA	normal	703228.87	+72.60%	26534.99
SEDA	underthreaded	807288.27	+84.27%	31984.25

Figure 4.25: Site search engine: L2 instruction cache misses (top)



The results of this experiment are more equivocal than those of the image processing server experiment: they still suggest that the thread-per-core scheduling policies are superior to SEDA thread pool-per-stage policies, but by a much smaller margin (1%) on the primary metric, response rates (tables 4.9 and 4.10 and figures 4.17, 4.18, 4.19, and 4.20). The difference between these results and those of the previous experiment can be attributed to the presence of a single, extremely CPU-intensive bottleneck stage, the multi-term set combiner, among a large number of less intensive stages. This combination exposed some of the weaknesses of thread-per-core policies, the most significant of these being the general inability of existing thread-per-core policies to limit the CPU proportions dedicated to specific stages. The thread-per-core policies spend most of their time visiting the bottleneck stages, which allows queues to build up elsewhere. (This pattern of queue

Figure 4.26: Site search engine: L2 instruction cache misses (bottom)



lengths was observable in per-stage statistics, not shown.) Thread pool-per-stage policies, in contrast, have the advantage of imposing automatic time limits on stage visits, i.e. thread quanta enforced by the operating system. The latter point suggests a solution to the issue for thread-per-core policies, namely limiting the number of events processed on each visit (a gated service regime) and/or interrupting visits when another stage's queue length that exceeds a threshold, as in [LP02]. However, previous research and my own testing indicates that these techniques are difficult to implement in a general way, due to the difficulty of fixing static gate and/or queue length thresholds for any given application. When thresholds are set too high the gated service regime is reduced to the exhaustive service regime tested here. When thresholds are set too low events must wait longer to be processed, and performance is sacrificed for fairness and guaranteed progress.

Table 4.11 and figures 4.21 and 4.22 show mean response times of the different servers. Thread-per-core policies perform well here, with response times in the second range. The notable exception to this rule is the DBR policy, which is clearly a non-starter by any metric. Constantly switching to the bottleneck stage (the multi-term set combiner) results in extremely high response times.

Generally speaking, thread-per-core policies such as MG1 tend to suffer when there are many stages that are mostly idle, as in the benchmark server. The non-negligible setup cost of visiting idle stages means that busy stages must wait longer for service. Experience with other servers has indicated that polling thread-per-core policies have visit success rates on the order of 5-10% in a hot system, which implies they are spending a non-negligible amount of cycles visiting mostly-idle stages. Here the Color policy and all thread pool-per-stage policies clearly benefit from being "demand-driven" and never visiting idle stages, and this is reflected in the observed response times.

The thread-per-core policies (excepting DBR) were more data cache-friendly than SEDA configurations (figures 4.23 and 4.24). The policies with the poorest throughput (DBR and SEDA overthreaded) exhibited the best L2 instruction cache locality (figures 4.25

and 4.26), but this is obviously a red herring. Of the servers with good throughput the ones with thread-per-core policies had fewer L2 instruction cache misses, as expected.

4.4 Experiment: comparing load balancing strategies in a site search engine

The final experiment evaluates the efficacy of the two load balancing strategies described in chapter 3 in terms of the throughput of thread-per-core-scheduled servers. The workload, client and server software, metrics, and environment are the same as in the previous experiment. One thread-per-core scheduling policy (Cohort/wavefront) with "normal" stages is evaluated with different load balancing strategies: no load balancing, i.e. the setup of the previous experiment; replicating the stage graph 2, 4, and 8 times; and partitioning the stage graph into 2 and 4 partitions.

Besides throughput (mean and max response rates) the primary metrics of interest here are the number of data and instruction cache misses of the servers employing the various load balancing configurations. The hypothesis is that load balancing can reduce cache misses by limiting the number of cores that participate in all (replication) or part (partitioning) of a request's processing cycle, thus increasing per-core locality while decreasing inter-core cache traffic. From this perspective replicating the stage graph on each core may be considered an ideal load balancing strategy, since it imitates the ideal Cohort scheduling scenario of a single core "moving" a batch of requests through the stage graph. However, on a multiprocessor system the advantages of this or any other load balancing strategy must be weighed against the real danger of load imbalance between processors, where one replica (processor(s)) is heavily loaded while others are mostly idle, resulting in lower throughput despite better cache behavior.

The best load balancing strategy, then, is one that strikes the right balance between good per-core performance and the performance of the system as a whole. Partitioning makes this tradeoff explicit by assigning stages to partitions according to their projected load, at the expense of more cores touching every request. Replication, on the other hand, relies on the implicit assumption that simple, non-clairvoyant distribution of requests to replicas (e.g. via round-robin) will not induce significant load imbalances between replicas of any granularity. Both types of load balancing assume that arrival rate and service time distributions for requests do not change drastically on short notice.

4.4.1 Results

The results of the load balancing experiment are shown in tables 4.14, 4.15, 4.16, 4.17, and 4.18 as well as boxplot figures (one per metric, with each box representing ten runs) with the same data.

Figures 4.27 and 4.28 plot the response rates for the different server configurations. They indicate that there is a significant advantage to replicating a stage group, particularly in having one replica for each of the 8 available cores. The latter configuration produced mean response rates more than 8% higher than the baseline server without load balancing. Somewhat surprisingly, the same configurations also had lower response times (table 4.16

Table 4.14: Site search engine with load balancing: mean response rate (responses/s)

Policy	Configuration	Mean	Difference in mean	Standard deviation
Cohort	8 replicas	922.60		1.89
Cohort	4 replicas	911.62	-1.20%	3.67
Cohort	2 replicas	892.93	-3.27%	3.74
Cohort	2 partitions	873.73	-5.44%	4.29
Cohort		846.46	-8.61%	12.49
Cohort	4 partitions	656.20	-33.75%	55.36

Figure 4.27: Site search engine with load balancing: mean response rate

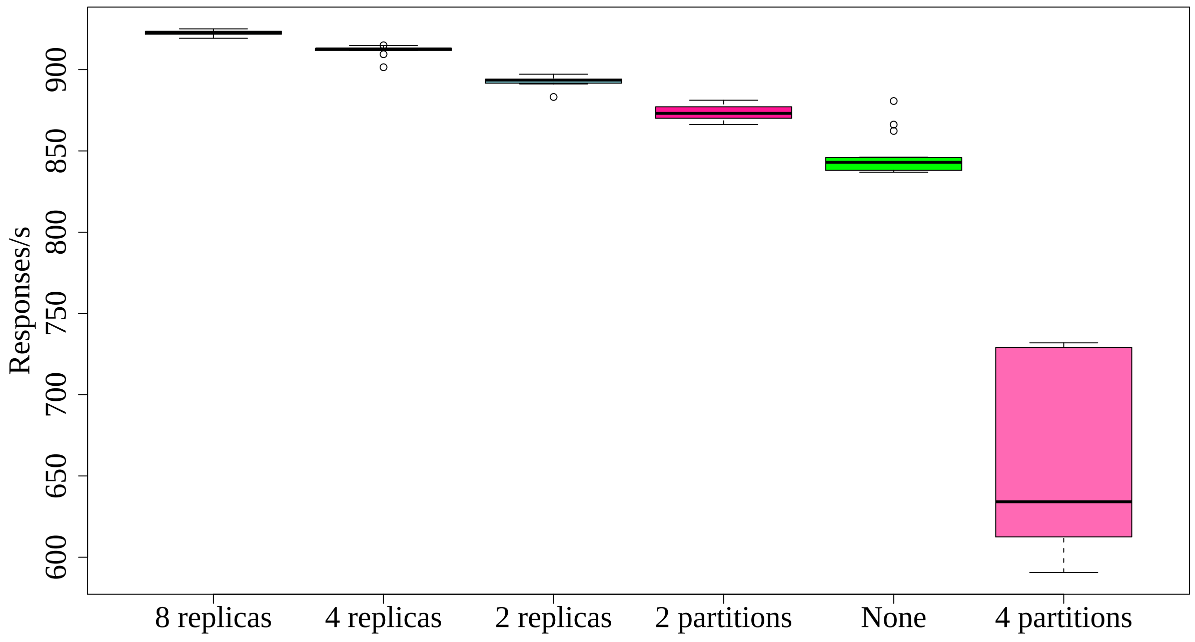


Table 4.15: Site search engine with load balancing: max response rate (responses/s)

Policy	Configuration	Mean	Difference in mean	Standard deviation
Cohort	8 replicas	1433.53		27.12
Cohort	4 replicas	1423.26	-0.72%	15.89
Cohort	2 replicas	1418.20	-1.08%	9.58
Cohort	4 partitions	1415.46	-1.27%	187.50
Cohort	2 partitions	1383.28	-3.57%	21.21
Cohort		1381.92	-3.67%	20.17

and 4.29). This is likely due to smaller batch sizes in the individual replicas/partitions, which resulted in lower queueing times.

Of the two load balancing strategies proposed in chapter 3, replicating the stage graph fared decidedly better as a general strategy than partitioning. The potential problem of load imbalances between cores was a non-issue for the benchmark server, where a simple round-robin distribution of requests to replicas sufficed to keep all cores active. The partitioning strategy, which is designed to explicitly address this problem at the expense of some performance, thus proved to be unnecessarily pessimistic.

Figure 4.28: Site search engine with load balancing: max response rate

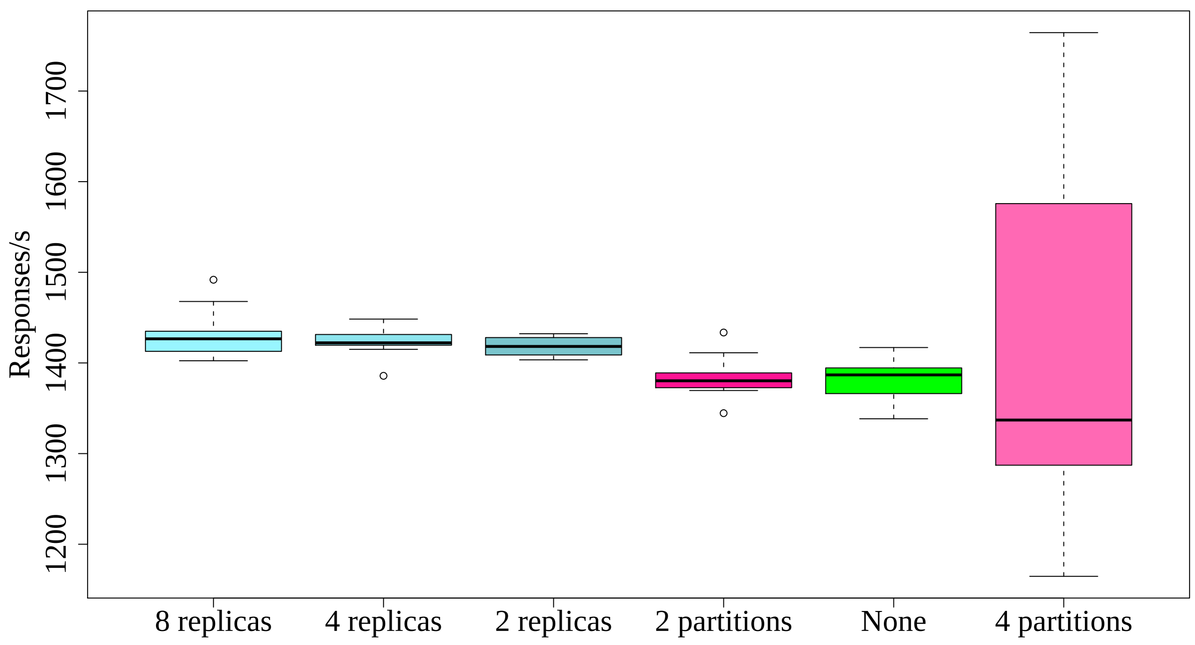


Table 4.16: Site search engine with load balancing: response time (ms)

Policy	Configuration	Mean	Difference in mean	Standard deviation
Cohort	8 replicas	111.57		6.01
Cohort	4 replicas	241.18	+73.49%	15.10
Cohort	2 replicas	541.13	+131.63%	31.93
Cohort	2 partitions	991.70	+159.55%	88.66
Cohort		1351.58	+169.50%	251.16
Cohort	4 partitions	4761.27	+190.84%	884.03

Table 4.17: Site search engine with load balancing: L1 data cache misses

Policy	Configuration	Mean	Difference in mean	Standard deviation
Cohort	4 partitions	2424127.00		403818.14
Cohort	8 replicas	3308136.78	+30.84%	338371.36
Cohort	2 partitions	3507673.00	+36.53%	567201.65
Cohort	4 replicas	3800302.40	+44.22%	499706.90
Cohort	2 replicas	3829168.60	+44.94%	330109.05
Cohort		3959960.00	+48.11%	394522.74

The majority of the load-balanced servers also exhibited greater data cache (table 4.17 and figure 4.30) and instruction cache (table 4.18 and figure 4.31) locality than the server without load balancing, albeit to varying degrees. This is evidence that load balancing does have the intended effect of improving cache behavior. On the other hand, it also is clear from the results that the improvement does not automatically translate to better throughput, since poor load balancing can still leave some cores more idle than they would be with no load balancing (table 4.14 and figure 4.27). Although the 2-way partitioned servers fared slightly better in terms of throughput than the server without load

Figure 4.29: Site search engine with load balancing: response time

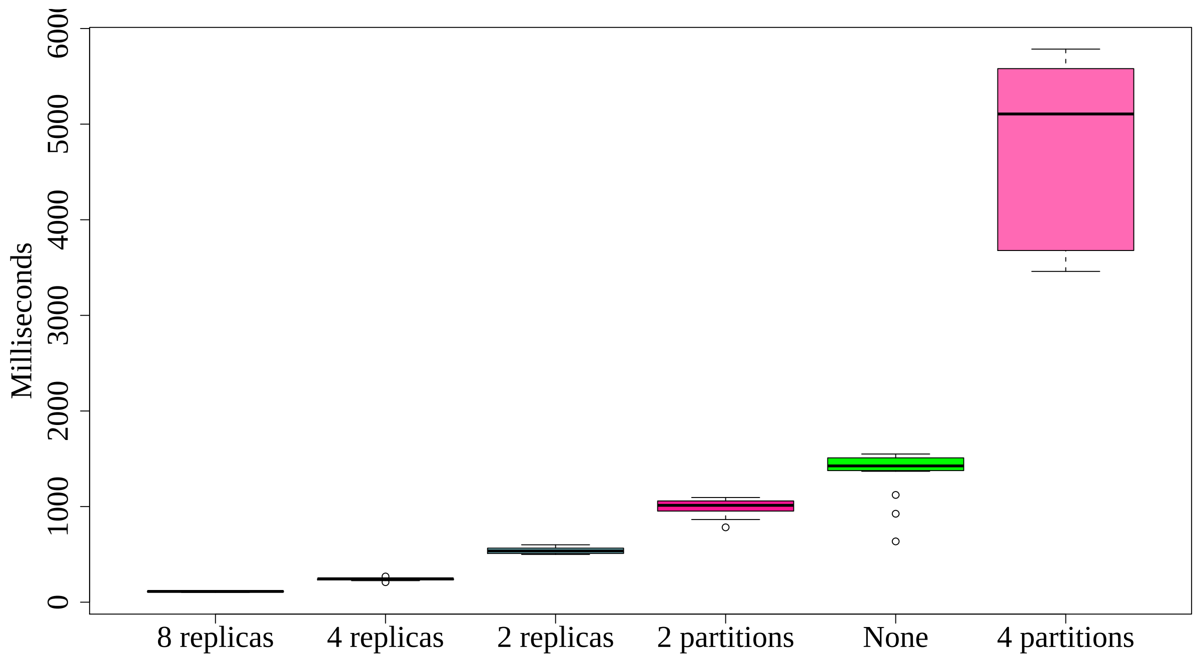
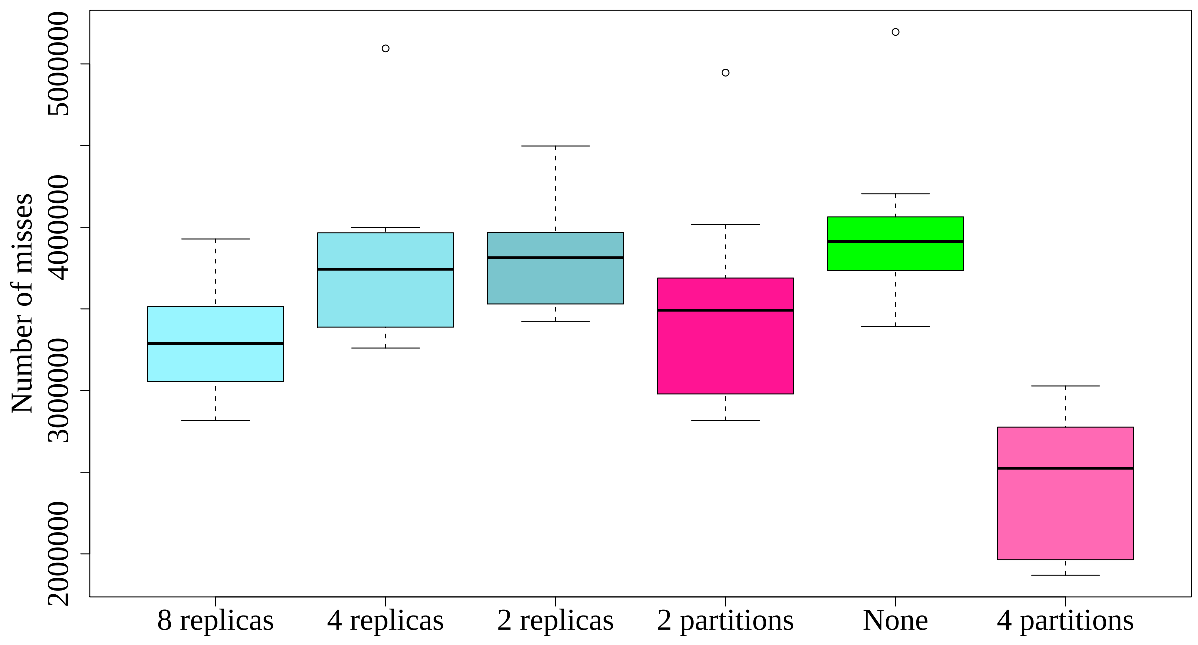


Figure 4.30: Site search engine with load balancing: L1 data cache misses

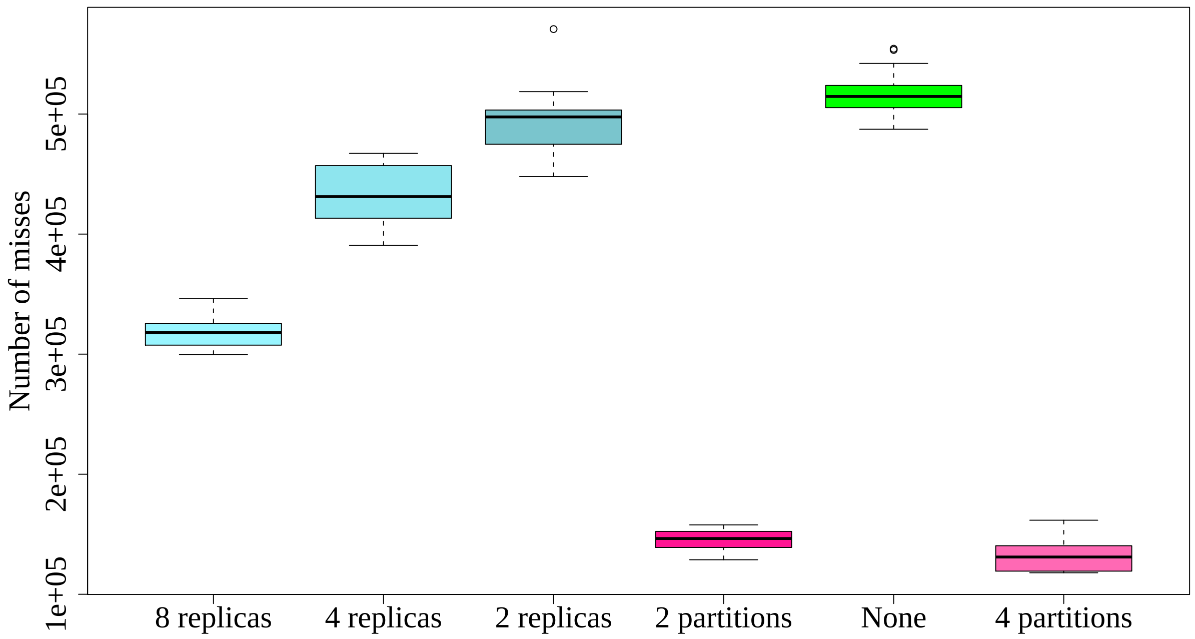


balancing, the 4-way partitioned servers were noticeably worse off. This was due to an imbalanced partition that occupied two cores while leaving the others mostly idle. The risk of imbalances between partitions increases with the number of partitions; a server with an 8-way stage graph partition was not included in the experiment, since it was clear from the outset that CPU-intensive stages such as the multi-term set combiner would not be able to keep up with the total load of the server if they were limited to running on very few cores.

Table 4.18: Site search engine with load balancing: L2 instruction cache misses

Policy	Configuration	Mean	Difference in mean	Standard deviation
Cohort	4 partitions	133624.00		13428.51
Cohort	2 partitions	145412.33	+8.45%	8021.85
Cohort	8 replicas	318811.00	+81.86%	13184.99
Cohort	4 replicas	433259.10	+105.71%	24562.92
Cohort	2 replicas	497039.40	+115.25%	31200.64
Cohort		517747.13	+117.94%	18749.74

Figure 4.31: Site search engine with load balancing: L2 instruction cache misses



4.5 Summary

Over the course of my PhD studies I have implemented a number of servers in C++, all of which rely on a common code base called Yield. In this chapter I evaluated the performance of two of these servers in three experiments: the first experiment comparing the performance of staged concurrency and thread-per-connection concurrency as well as various stage scheduling policies in the context of an image processing application; the second comparing the throughput of stage scheduling policies in a site search engine; and the third comparing the effectiveness of load balancing strategies for the same server. The results of the experiments support a number of conclusions:

- Staged concurrency outperforms thread-per-connection concurrency on throughput-related metrics, though because of its fairness properties thread-per-connection concurrency often exhibits lower response times.

- Thread-per-core stage scheduling policies as a whole are superior to thread pool-per-stage policies in terms of throughput on CPU-intensive benchmarks.

- The thread-per-core policies as a class are comparable to each other in terms of the same metrics.
- Load balancing stages in a server with thread-per-core scheduling can increase the throughput of the server.
- Of the two load balancing strategies proposed in chapter 3, replicating the stage graph is the more effective in increasing server throughput.

The next chapter will outline some of the qualitative lessons I have learned in developing these and other servers.

Chapter 5

Experiences with staged servers

Over the last six years I have implemented and experimented with many different types of staged servers, attempting to find suitable benchmarks for Yield. Most of these attempts were unsuccessful, though they were all instructive. The failed benchmark servers fell into two main categories: those implemented written primarily in Python for the Python back end of Yield and those implemented entirely in C++.

5.1 Python-based servers

Dynamically-typed languages such as Python, Ruby[1], and Perl[2] have become the tool of choice for increasing numbers of server developers. This choice is often motivated by the need to develop, deploy, and alter server applications rapidly with minimal knowledge of the language. Like other programmers I was initially attracted to dynamic languages – in my case Python – by the ease with which I could write and test simple servers.

In order to bring the strengths of Python to Yield I embedded a Python interpreter as a Yield event handler. The main task of this event handler is to translate events sent to it in C++ into a form that can be processed by Python, and to translate responses and other events coming out of Python back into C++ for consumption by other event handlers. Handling events from C++ on the Python side also called for some adaptation, in order to take advantage of Python's built-in type introspection, provide convenience methods for programmers, and other niceties.

The major disadvantage of embedding Python in a environment that is designed for high performance is the fact that the reference Python interpreter relies on a process-wide Global Interpreter Lock (GIL) to serialize access to the interpreter's state, which makes it effectively single-threaded. The same limitation applies to several other dynamic language interpreters, including those for Ruby and PHP. Multiple threads can enter the Python interpreter, but only one can execute normal Python operations (such as string manipulation or regular expressions) at a time. This obviously limits the amount of parallelism a multi-threaded Python program can exploit. For this reason as well as the general difficulty of writing multithreaded programs the vast majority of Python code is single-threaded. The exceptions are multithreaded servers that block on socket operations

[1]`http://www.ruby-lang.org/`

[2]`http://www.perl.org/`

such as `send` and `recv`. In these servers the C interface between Python and the associated C functions (called a Python "extension") voluntarily releases the GIL before going into a blocking operation, so that another thread can run in the interpreter while the first thread is blocked on network I/O.

The serialization of most Python operations implies that the Python interpreter cannot block waiting for a response to an event it has sent into C++. Instead it must save the relevant Python state before an event is sent and then restore the state after the response has been received. The reference Python implementation provided no means for doing this automatically, which means that event-driven servers using the reference implementation must employ manual stack management.

Fortunately, there is an alternative implementation of the Python interpreter, called Stackless Python[3], that allows programmers to create many automatically-managed stack contexts within a single-threaded Python program. It accomplishes this by storing all Python runtime state on the C heap instead of the stack. This allows the Stackless interpreter to simply switch C stack pointers to restore a given Python stack. The developers of Stackless Python have also created an extension to the standard CPython interpreter that does the same job, albeit in a less efficient manner. Yield's embedded Python interpreter will link to Stackless Python if it is present and emulate the Stackless API with the extension if not. The automatic stack management provided by Stackless allows Yield/Python servers to be completely agnostic of any and all event passing "under the hood". The program simply makes an interface-defined call and receives the return value, as if there were no event passing. The machinery below the call creates an event with the parameters of the call, sends it through the Python to C++ gateway, then switches the Python stack back to another that might be runnable (because of a received response, a long-running computation, etc.).

5.1.1 Web-based chat server (2006)

My initial foray into Python servers was a web-based chat server, which was inspired by the chat server deployed at `wie-ich.de`. The original implementation on `wie-ich.de` was written with Java servlets and streamed chat messages to web browsers via a neverending HTTP chunked response. My Python implementation was based on a design by Felix Hupfeld in which web browsers poll the HTTP server for updates to a chat channel. The poll request was a Remote Procedure Call from the browser that specified the last update the browser had seen in the channel, as a logical timestamp, with the ability to specify multiple channels per call (e.g. multiple group and private chats and a channel for control messages). The set of logical timestamps, one for each channel, was treated as a version vector [DSPPR$^+$86].

The polling design has the advantage over the more conventional streaming design of allowing clients to easily recover the state (messages) of a channel after a voluntary or involuntary disconnect. The major disadvantage of this approach is that polling puts more load on the server. Another problem with this implementation was that the server had to check the data structures for each channel each time a client polled and serialize all chat operations into a form the web browser can understand, in this case the JavaScript Object Notation (JSON). Since the server was mostly implemented in Python, this meant that

[3] `http://www.stackless.com/`

the single-threaded Python stage was a major bottleneck. This proved to be a recurring (if not entirely unexpected) problem with Python servers. The solution was to rewrite performance-critical and thread-safe parts of the code such as e.g. JSON serialization in C++, wrap it in a separate event handler with a separate stage, and drive this stage via event-passing from the Python interpreter. Unfortunately, alleviating one bottleneck in Python tends to expose another in the same code, which hides another, and so on, until the programmer has spent more time exporting code into C++ than he would have simply writing the server in thread-safe C++ from the beginning.

5.1.2 HTTP file server (2007)

Serving static content from disk or memory is still one of the primary functions of HTTP servers, and static file benchmarks are still a touchstone for HTTP server performance. I have benchmarked HTTP file servers several times in the course of developing Yield, notably in my Master's thesis [Gor05], where I compared Yield's static file serving performance (in C++ only) to that of the original SEDA implementation (in Java) on a SPECweb99-like workload. As a follow-on experiment for this dissertation I decided to write a hybrid Python/C++ HTTP server and benchmark it against some real-world servers implemented in C, as well as a few implemented in pure Python. The purpose of this experiment was twofold:

1. to demonstrate that adding C++ stages to a Python server (one in which all unique code for the server is written in Python) can compensate for some of the performance limitations of Python, such as frequent memory copying
2. to show that the same hybrid server can be competitive with C servers, at least until the single-threaded Python interpreter occupies an entire core

Workload

The workload for this benchmark was derived from the static file part of SPECweb99 [Sta99]. The data set consisted of 13 gigabytes of files with file sizes ranging from less than 1 kilobyte to around 9 megabytes. File accesses assumed a Zipf distribution, with small files requested much more frequently than large files. No part of the official SPECweb99 distribution (file set generator, client, etc.) was employed in this benchmark. Most of the implementation of the benchmark client and workload were reworked from my previous benchmarks of Yield vs. the Java implementation of SEDA for my Master's thesis. That setup was in turn adapted from the SPECweb99-like benchmark client written by the author of SEDA [Wel02].

Client software

For this experiment I used `httperf` session logs to describe the SPECweb99-like workload. During a benchmark run four instances of httperf on the same client machine replayed unique (per-instance) 1000-session logs against the server. The logs were generated offline according to the SPECweb99 rules. For this experiment I also used the inter-request think times of Pariag et al. [PBH$^+$07], who derived the following times based on previous analyses of user behavior: 3.0 seconds for the "inactive" think time that users spend deciding on which link to follow and 0.343 seconds for the network and client-side latency between "active" requests. The latter time represents, for example, the latency between a web browser's reception of a site's index page and the time the server receives the first request for an element of the index page, such as an image.

Server software

The server software for this experiment included three production-grade servers in C (`lighttpd 1.4.18`, `nginx 0.6.25`, and Apache 2.2.8), two pure Python servers (Medusa 0.5.4 and Twisted 8.0.0), and a C++/Python Yield hybrid. The servers vary in complexity and robustness, but they are comparable in terms of their basic operation. In order to ensure that this was the case I disabled many non-essential features of all the servers, including logging and bookkeeping facilities. I also disabled user-level file caching on servers that supported it, since not all servers were capable of caching files in userspace. Because primary storage was large enough to fit the most frequently-requested files clever user-level caching would not have bought much, in any case, and the operating system caching was sufficient.

The following sections describe the HTTP file servers in detail:

- `lighttpd`[4] and `nginx`[5] are classic single-threaded, non-blocking event loop servers with optional Flash-style worker pools for offsetting blocking system calls when asynchronous disk I/O primitives are not available. Both servers go to great lengths to avoid blocking by e.g. advising the kernel on which memory pages the server will send to the network next so that the kernel can pre-fetch them from disk if necessary. As of this writing approximately 0.89 percent of web servers on the Internet run `lighttpd`, according to a recent Netcraft survey[6]. Both servers were used in their default configurations.

- **Apache**[7] is a classic kernel thread-per-connection web server and the most widely deployed web server on the Internet, with 49.1 percent market share, according to the same Netcraft survey. Apache offers a variety of knobs and dials for tuning the maximum and minimum number of threads, keep-alive timeouts, etc. as well as allowing administrators to choose at compile time whether threads or heavyweight processes should be used. For this experiment I measured the performance of an Apache server configured to use a maximum of 640 threads, a parameter that I copied from a site that serves many static files.

- **Medusa**[8] is a pure Python server with a single-threaded event loop. The server is started programatically from a Python script rather than bootstrapped from a configuration file. A startup script for a basic HTTP file server is included in the Medusa distribution. For the experiment I modified the script to disable logging but left it otherwise unchanged.

[4]`http://www.lighttpd.net/`
[5]`http://nginx.net/`
[6]`http://news.netcraft.com/archives/web_server_survey.html`
[7]`http://httpd.apache.org/`
[8]`http://www.nightmare.com/medusa/`

- **Twisted**[9] is a high-level networking framework in pure Python. Like Medusa it also relies on a non-blocking event loop, though unlike Medusa, which uses `select` exclusively, Twisted can take advantage of efficient event notification primitives such as `epoll` on Linux. To the best of my knowledge Twisted has never been subjected to a rigorous evaluation of any kind, which is noteworthy in light of the fact that the entire design of the framework, with event loops and continuations, was motivated by the desire to avoid blocking, ostensibly as a means of achieving high performance. Like Medusa, Twisted is driven by a Python script rather than a configuration file. The script for this benchmark was adapted from an HTTP file server example in the Twisted documentation.

- **Yield** hosted an HTTP file server that consisted of three stages in a thread pool-per-stage group: a single-threaded I/O stage that accepted connections and read and wrote data from and to the network; a single-threaded Python stage that received requests from the I/O stage, serialized them into Python, translated request URLs to file paths (in Python), and sent requests to the disk I/O stage for the files to be read; and a multithreaded disk I/O stage that `read` the files from the file system and sent the contents back to Python, where they were attached to an HTTP response and sent on to the I/O stage.

Metrics

Response rates with response timeouts are the most widely-accepted metric of HTTP file server performance [MJ98, PBH+07, WCB01, vBCZ+03]. `httperf` measures response rates by sampling the number of on-time responses over five second periods. At the end of an experiment run `httperf` displays the average, minimum, and maximum of these per-period response rates. The average was taken as the primary metric for a given experiment run.

For this experiment the time limit for valid responses was set to five seconds, as measured from the the point at which the client sent the entire request to the server (i.e. when the `writev` system call completed) and the point at which the client received the first byte of the response (from the `recv` system call). Five seconds is a lower bound suggested by the `httperf` man page[10]. Other authors have used response time limits as high as fifteen seconds [PBH+07], though intuition suggests that few users with fast connections would be willing to wait fifteen seconds for a nine megabyte file when network proximity should allow a server to deliver the file much faster than that.

The exact choice of a response time limit becomes relevant only when a server is under heavy load or overload and is unable to write responses to the network on time. The C servers (`lighttpd`, `nginx`) were never overloaded in this experiment and always delivered responses in well under five seconds; this can be inferred from the near-linear curve of request vs. response rates in the figure below. The pure Python servers, in contrast, were overloaded fairly early in the set of experiment runs. Increasing the response timeout to 15 seconds or more would have shifted the peaks in the Python server curves slightly, but the overall pattern of the graph would have remained the same.

[9] `http://twistedmatrix.com/`

[10] `http://www.hpl.hp.com/research/linux/httperf/httperf-man.txt`

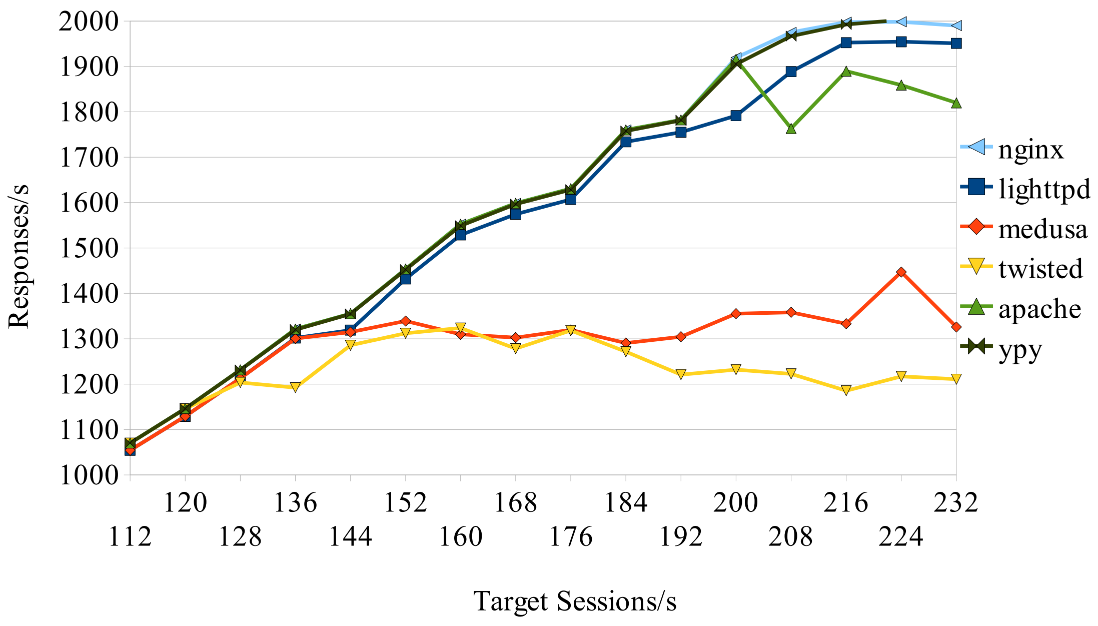
Environment

The operating environment for this experiment was the Alibaba cluster described in section 4.3.5. Both the client and server were run on cluster nodes, with one node for each. The client worker node hosted four instances of `httperf` (corresponding to the four cores of the machine), each with a unique session log.

Results

Figure 5.1 is a graph of the mean response rates of the different servers, with increasing session/connection initiation rates on the x-axis and the rate of on-time responses on the y-axis. (The rate at which new sessions are initiated is the same as that for new connections, since only one session was replayed per connection in this experiment.)

Figure 5.1: HTTP file servers: mean response rates



The session rates delimited on the x-axis are target rates rather than actual rates, which differed between servers; since `httperf` works in an open loop it cannot consistently guarantee a specific connection rate across all experiment runs. Furthermore, httperf will close a session/connection if any request in the session times out, so that new sessions/connections will still be initiated for overloaded servers. This is necessary to continue increasing the session rate in an open loop even when the server is overloaded. Each point in the data series represents a single experiment run, with each of the four client instances replaying its entire 1000-session log at a given session initiation rate.

Note that figure 5.1 does not include the extremes of low and high session rates. At low session rates all of the servers performed equally well. At high request rates the curve of the Python servers and Apache continued in the same zig-zag pattern of peak and sub-peak, while the response rates of the C servers continued to increase (albeit at a less dramatic rate) until the clients were overloaded (i.e. unable to initiate enough sessions quickly enough without saturating the client machine's CPU and/or running out of file descriptors). With more client machines the latter servers could have been pushed into overload, but doing so was not necessary to demonstrate the differences between the pure Python, C++/Python hybrid, and pure C HTTP file servers.

As expected, the C++/Python Yield hybrid is competitive with the pure C servers until the single-threaded Python stage becomes CPU-bound. The CPU consumption of the Python stage dwarfed that of the other stages (for network and disk I/O). The performance of the pure Python servers was even more severely limited by the single-threaded Python interpreter. Somewhat surprisingly, Twisted fared significantly worse than Medusa, despite the former's use of `epoll` in preference to `select` and its reputation as the Python networking framework of choice for high-performance servers.

5.1.3 RUBiS server (2007)

The Rice University Bidding System (RUBiS)[11] is a benchmark for database-backed web applications. The benchmark setup is modeled after popular online auction sites such as eBay, where users browse, buy, and sell items. Information on items, item categories, users, and bids is stored in a database, which is also covered by the benchmark. RUBiS was designed for the purpose of comparing application servers in combination with relational databases [CCE+03]. In the reference implementation of RUBiS only one database (MySQL[12]) is supported, while the application server is varied in order to measure the performance of different implementations of the auction prototype: J2EE, servlets, servlets and J2EE, and PHP.

The primary purpose of this experiment was to demonstrate that a Yield-based implementation of the benchmark server can achieve higher throughput than other implementations of the same server, even when the web server is not CPU-bound. Many real-world web applications rely on databases to store frequently-updated application data such as user histories, product stocks, etc. These applications are typically bound to the performance of the database on the back end, and the performance of the web application server on the front end is much less critical to the throughput of the application as a whole. Nevertheless, having an efficient front end can still be important when an application has to scale to multiple machines. Better single node performance means that fewer front end machines are needed, which reduces initial hardware costs as well as the running costs of power, maintenance, and administrator time. The performance of the front end was the focus in this experiment.

The secondary purpose of the experiment was to show that a Python implementation of the benchmark application that is hosted in Yield can outperform the multithreaded reference Java implementation of the application, despite having significantly fewer lines of code and running on a far less optimized virtual machine. The Java servlets implementation of RUBiS was the starting point for my implementation for Yield, which was transliterated and adapted servlet-by-servlet into Python code that could run in Yield's back end Python interpreter. All unique code for the Yield implementation was in Python, though like the HTTP file server the Python implementation of RUBiS also exported some non-unique code into C++ in order to achieve greater efficiency and concurrency. In this case the C++ code consisted of an asynchronous database client interface I had developed

[11]`http://rubis.objectweb.org/`
[12]`http://www.mysql.com/`

previously and a generic wrapper for the `ctemplate`[13] template library that formatted objects from Python as HTML. The database client interface created one stage per database connection, with each stage's event handler executing SQL statements and fetching results on/from that connection. This allowed the Python interpreter to avoid blocking on database client API calls, which are typically implemented in a blocking manner. Exporting HTML formatting code into C++ mitigated a serialization/copying bottleneck in Python.

The resulting Python implementation of the RUBiS web application consisted of only 890 lines of Python code and 634 lines of HTML templates = 1,524 lines of code specific to this application, while the functionally equivalent Java servlets implementation of RUBiS weighed in at 4,797 lines (including inline HTML), as measured by SLOCCount[14]. The smaller code base and separation of application logic from presentation made the Python implementation easy to develop and the resulting code remarkably clean.

Unfortunately, experiments comparing the two implementations of RUBiS were largely unsuccessful. The throughput of the staged Python/C++ hybrid implementation was in fact higher than that of its thread-per-connection competitor in Java, but the differences could be explained by the extremely poor quality of the design and code of the Java implementation rather than any essential advantage of the Python/C++ version. In particular, the Java servlets used far too many database connections, which resulted in unnecessary database lock contention that artifically limited throughput when the database became CPU-bound. The Java thread-per-connection server quickly expended its threads blocking on the database so that there were no threads available to process new connections, which in turn led to client connection timeouts.

5.1.4 WebDAV server (2007)

As a complement to the Python HTTP file server described in section 5.1.2 I also designed and implemented a Python-based Web Distributed Authoring And Versioning (WebDAV) [Dus07] server. Like many other types of servers on the Internet and unlike read-only HTTP file servers, WebDAV servers are widely deployed but rarely benchmarked. In fact, I was unable to find any benchmarks for WebDAV whatsoever, although the protocol has been in production use for almost a decade. This is likely due to the fact that WebDAV servers are seldom placed under much stress: as the name suggests, they are primarily used for authoring web sites, as a practical read/write extension to the read-only HTTP used by browsers. This means that the majority of installations do not need to scale to more than a dozen users. However, WebDAV interfaces are also provided by a number of large online file storage and sharing services, such as Apple's MobileMe[15] and box.net[16].

In its essence WebDAV is similar to NFS and other mostly-stateless network file systems in that it uses Remote Procedure Calls as stand-ins for POSIX-like local file operations (read, write, lock, et al.). The main difference is that WebDAV uses HTTP verbs (GET, PUT, et al.), URLs, and XML request bodies to indicate operations and their parameters rather than binary RPCs. Although a network file system would seem an unlikely candidate for evaluating CPU-bound scheduling algorithms, the inclusion of XML changes the

[13]`http://code.google.com/p/google-ctemplate/`
[14]`http://www.dwheeler.com/sloccount/`
[15]`http://www.apple.com/mobileme/`
[16]`http://www.box.net/`

situation. XML parsers and generators are quite CPU-intensive, thanks to the excessive string copying that is required to build a DOM with decoded, null-terminated tag names and string concatenation in outputting validated XML.

In the absence of any known workloads I had to design and implement one of my own, starting from representative traces. After a long search for such traces a system administrator at the University Library in Cambridge volunteered to share WebDAV logs from one of the library's servers. The logs were from a server that librarians across the University and Colleges use to edit the metadata for small collections[17]. The metadata for a collection is stored in a single XML file, with the size per file ranging from 63 bytes to 23 megabytes with a mean of 170K and a median of 12K, as measured at the time of a request. The logs spanned the period from August, 2003 to March, 2008, approximately 70,000 requests from a few dozen users. The server is typical of WebDAV deployments in that it sees very low demand, but atypical in having so many different users editing only one or a few files each.

In order to generate a benchmark workload from the logs I first constructed a Markov model of user sessions, which described the probable sequences of request types (PROPFIND, GET, PUT, etc.) for a session while accounting for certain nuances of the WebDAV protocol and idiosyncrasies of the most common client, the one included in Windows. I then used this description to generate synthetic session logs for `httperf`.

Unfortunately, the choice of `httperf` was to be my undoing: the client was simply not designed for read-write benchmarking. It could not handle uploading large bodies or most of the WebDAV-specific HTTP request methods. After numerous modifications and attempts to cajole `httperf` into producing enough traffic quickly enough to put significant load on the server, I had to give up on this benchmark and move on.

5.1.5 SMTP server (2007)

My next attempt at a CPU-intensive benchmark was an SMTP server with a built-in spam filter. This project was inspired by two papers that analyzed server logs of a large ISP [Cla04, Cla05]. In his analysis Clayton noted that the the majority of the SMTP deliveries received by the ISP were eventually marked as spam, and the combined size of spam messages far outweighed that of legitimate email. Due to this imbalance most large mail server setups are bound to the speed of their spam filters rather than the speed of disk I/O. The spam filters in turn are bound to the CPU of the hardware on which they are running.

Some of the most popular spam filters are written in dynamic languages such as Perl and Python, though not because the languages are dynamic, per se, but because they tend to have well-integrated and extensive string processing facilities. Tighter integration between mail servers and spam filters has been advocated for several years [PRH07], and an SMTP server with a built-in, CPU-intensive spam filter sounded like an ideal benchmark for Yield.

There were a number of stumbling blocks in this plan. SMTP servers are critical to the Internet infrastructure, yet they are rarely benchmarked publicly with realistic workloads. There is a SPEC-endorsed benchmark for combined POP3/IMAP/SMTP mail servers,

[17] `janus.lib.cam.ac.uk`

SPECmail2001 [Sta01], which is similar in spirit to SPECweb96 [Sta96]. The SPECmail2001 benchmark workload was based on a statistical analysis of mail server logs by Bertolotti and Calzarossa [BC01]. Unfortunately, the benchmark was designed with only legitimate emails in mind, and the patterns observed by Calzarossa in the late 1990s and early 2000s are no longer representative of today's mail server traffic. In the absence of any other serious mail server benchmarks I had to write my own. This was relatively straightforward: I used a portion of the TREC spam corpus[18] as a data set and wrote an SMTP client in Python that initiated new SMTP connections to the server with an exponential arrival rate, taking guidance from a paper on spam-infected mail workloads [GCA+07]. The benchmark client worked well, kept accurate statistics, and was not CPU-bound.

Rather than implement the entire server in Python and hitting another bottleneck, I chose to embed another language, Lua[19], for the sole purpose of spam filtering in a multithreaded Lua stage. Lua is well-regarded for its simplicity – it only has one data structure, an associative array – and the ease in which it can be embedded. More importantly, there is a thread-safe C extension to Lua that implements the Orthogonal Sparse Bigrams with confidence Factor (OSBF) [SACY04] spam classification algorithm, which won the TREC spam track in 2006. After porting the Unix-specific parts of the OSBF back end (i.e. the routines used to store classifier rules in files) to Yield's platform library I trained the algorithm using part of the TREC spam corpus. Then I wrapped the extension in machinery to translate classification requests and responses to and from Lua. The main SMTP server state machine was implemented in Python, with raw data buffers shuttled between the C++ I/O front end and Python on the back end.

Despite the extra effort the mail server, like its predecessor the chat server, was also bound to Python – a single line in Python, in fact. The single line concatenated two buffers that had been read separately from the network. After various attempts to circumvent this line and others like it, I finally gave up on the idea of implementing performance-critical code in Python, its aptitude for string processing notwithstanding. In subsequent projects I restricted Python to the tasks of bootstrapping and coordinating CPU-intensive activities and left the latter to thread-safe C++.

One positive outcome of this attempt was a lesson in small optimizations. SMTP is a conversational protocol, with around 10 exchanges in a normal transaction, assuming connections are not re-used. This design makes it relatively easy for the server to schedule requests in Shortest Remaining Processing Time (SRPT) order, so that requests in the later exchanges are processed before initial handshakes, recipient specification, etc. Using this technique in Python I was able to increase the throughput of the server by approximately 30% in testing runs.

5.2 C++-based servers

5.2.1 C++ HTTP file server (2005)

For my Master's thesis in 2005 [Gor05] I explored some of the advantages and pitfalls of stage architectures for highly concurrent web servers and compared the performance of a

[18] `http://plg.uwaterloo.ca/~gvcormac/treccorpus/`

[19] `http://www.lua.org/`

staged HTTP file server using SEDA thread pool-per-stage concurrency to a thread-per-connection HTTP file server that utilized the same code base. I also benchmarked a C++ thread pool-per-stage HTTP file server against Welsh's Java HTTP file server, Haboob, from the original implementation of SEDA. For both benchmarks I used a SPECweb99-like workload and a setup similar to that used in the Python HTTP file server benchmark in section 5.1.2, except that I allowed file caching within the server since both the Java server and my own supported it. The performance of my thread pool-per-stage server was comparable to that of its Java counterpart and clearly superior to the thread-per-connection server. As in Welsh's work on SEDA, the crux of the benchmarks was admission and overload control rather than scheduling.

In my analysis of the benchmark results I pointed out several optimizations for staged servers, including aggressive object and buffer re-use; enforcing admission controls on new connections only rather than on every stage (vs. Haboob's per-stage controllers); responding to requests for smaller files before requests for larger files, similar to [CFHB99]; and minimizing the total number of stages in the system in order to decrease the number of runnable threads.

5.2.2 Cantag server (2007)

Cantag [RBH06] is a heavily-templated C++ library of algorithms for recognizing and deciphering black-and-white *tags* from images supplied by camera feeds in real-time. The tags encode some numeric identifier of an object, much like a bar code. They are used to identify *features* of an image, such as a person wearing one of the tags. Cantag was designed to allow application developers who need this recognition ability to compose pipelines of image processing algorithms that best suit the application. For example, there are specific algorithms for picking out the squares in an image, while other algorithms measure the circumferences of concentric circles in a tag. Like many image processing algorithms, the main recognition algorithms in Cantag are CPU- and memory-intensive. The Cantag algorithms are also highly modular, so they can be composed and work in parallel. This made Cantag seem like an ideal benchmarking apparatus.

I spent several days fitting the Cantag algorithms to Yield stages and adding a simple HTTP front end, which would be the target of e.g. cameras uploading images. Unfortunately, after benchmarking the server on a dual-core 3 GHz workstation and profiling and tweaking Cantag extensively it could barely process more than 1-2 highly artificial test images per second, far less than the 25 images per second required to keep up with camera feeds.

5.3 Lessons learned

Through the process of designing and implementing the staged servers described above I've reached a number of conclusions:

- **Single-threaded code usually costs more than its worth.** The single-threaded Python interpreter is a prime example of this. Python makes it easy to prototype a working server quickly. Moving beyond that is painful. It is better to simply discard the prototype and rewrite the entire server in a thread-safe setting than to try to retain parts of the single-threaded code. These will inevitably become bottlenecks. Caches are the main exceptions to this rule. Caches are usually single-threaded by necessity, the alternative being fine-grained locking, which would violate the design of stages as units of concurrency.

- **The more potential stages, the better.** The stage scheduler can combine two stages into a single scheduling unit by e.g. making one of the stages "pass-through". Stage boundaries should be delimited as early as possible rather than separated as needed. The latter approach usually takes more time than careful design does.
- **Stages should only have one function / process one type of event.** Limiting a stage to one function allows the stage machinery to accurately model the cost of processing events at a given stage, which in turn makes the stage graph more amenable to queueing theoretic analysis and optimization. The resulting stage graphs are usually acyclic.
- **Attach session state to events rather than having per-stage lookups.** Keeping session state (socket connections, open files, etc.) at the stage where it is used is a common beginner's mistake. This state is prone to memory/reference leaks. Looking up the session state at each stage is also unnecessary work.
- **Thread pool-per-stage is often good enough.** There is no real benefit to using a thread-per-core stage scheduler when the server is not CPU-intensive. SEDA thread pool-per-stage concurrency performs well enough for most servers and it is more tolerant of programmer laziness (e.g. unforeseen blocking that would have to be offloaded in a thread-per-core system) than other policies. Because threads are always associated with a specific stage it is also easier to profile.

Chapter 6

Related work

Multiplexing hardware resources such as processors and disks is one of the fundamental problems of Computer Science. The problem arises in many forms, from real-time scheduling for mission critical devices to scientific computing on huge clusters of machines. The range of algorithms and heuristics for solving computer scheduling problems is wider still, with techniques inspired by operations research, traffic engineering, production systems, and other fields. This chapter surveys some precedents for the research presented in this dissertation as well as related work in theory and practice.

6.1 Scheduling theory

As is often the case in Computer Science, there is a distinct divide between theory and practice of CPU scheduling. On one side there is a plethora of theoretical results on the stability, expected throughput, mean delay, and other performance characteristics of scheduling policies under certain system conditions. On the other side there is a mass of implementations that ignore theory entirely. That notwithstanding, for the discerning implementer scheduling theory can serve as a useful guide, though it can also obscure problems by introducing high-level abstractions to low-level implementations.

In addition to the queueing theoretic models described in section 3.1.1 on the MG1 scheduling policy, I considered several other types of models in my research on CPU scheduling in staged servers. Most of these models fall under the rubric of *production systems.*

Production systems

One of the most obvious physical analogies to staged servers such as the image processing server is that of a factory production line. Although production lines are conceptually equivalent to queueing systems [SZ92], researchers investigating production systems tend to focus on analyzing real-world systems rather than exploring the theoretical space. There are many different models for these real-world systems:

- *Push and pull production* are the classic modes of production lines [SZ92]. In a *push* production line work is admitted to the system and processed at the various stations according to a preset schedule. A *pull* production line, in contrast, only admits work to each station when the station explicitly signals its readiness for more work.

- *Goldratt's Theory of Constraints* is a well-known theory of production line management [Gol90]. In essence, the theory states that every production line has a bottleneck, called the *drum* or constraint, and the goal of production line optimizations is to ensure the drum is always supplied with work via a *buffer*. A *rope* mechanism signals the rest of the production line that the drum needs more work to do or has too much work.

- *Maximum pressure policies* are, practically speaking, a more mathematically rigorous formulation of the Theory of Constraints for general stochastic processing networks (e.g. multiclass queueing networks). To cite one example, Dai et al. describe some maximum pressure policy heuristics for server allocation that ensure there is always work for the bottleneck station [DL05]. These heuristics have the advantage of not requiring information on arrival rates, which is often difficult to gather in a real system. Instead they rely solely on the length of queues at a station and its successors in the network.

- A *job shop* is a more specialized model of a production system in which jobs can take alternative routes through the stations of the system. The counterpart of a job shop is a *flow shop*, where all jobs take the same route. A job shop is akin to a patient visiting multiple doctors at a hospital, where the the job "chooses" which stations it visits, while a flow shop more closely resembles a classic production line. A schedule for the job shop determines when specific jobs are dispatched to a station for processing, given some constraints such as setup times for switching a station from one job type to another. Various techniques have been applied to the problem of finding optimal schedules for different classes of job shops, where optimal is defined by criteria such as minimizing the total number of jobs, minimizing the number of late jobs, etc. The authors of [JR98] survey many of these techniques, from dynamic programming to expert systems to genetic algorithms.

General production system models are quite conceptually elegant, perhaps too elegant: their generality (and, in the case of the Theory of Constraints, vagueness) make them difficult to apply to computer scheduling problems without massive improvisation, which would negate much of the purpose of having a theory in the first place. The job shop model is slightly more precise, but also suffers from a mismatch in time and control granularity viz. the scheduling policies considered here. In a typical real-world job shop processing a single job may take minutes or hours, so an optimal schedule may be fixed for hours or days at a time. The cost of global synchronization within such a system is also relatively low compared to the gains of an efficient schedule. The sub-second granularity of multiprocessor scheduling, in contrast, makes global synchronization and global control prohibitively expensive.

Traffic models

Traffic systems are another conceptually attractive physical analog to scheduling in staged servers, and are similar to production systems in many respects [Hel05]. However, unlike most of the techniques for scheduling production systems, traffic control algorithms are often designed for decentralized operation. Systems for controlling traffic lights to optimize throughput [LH08] bear an obvious resemblance to polling systems, though controlling traffic lights usually involves timing constraints and optimizations that basic polling systems do not consider. For example, a traffic light may offer a green light to a direction when cars are either approaching or have left the intersection from that direction, while a polling system is usually work conserving in the sense that it will always switch away from an empty queue rather than spinning on it. Spinning may actually be a useful behavior in a large multiprocessor system, however, where waiting on a heavily-active, just-emptied queue instead of switching to another may increase throughput and decrease latency for the system as a whole. Unfortunately, in the absence of the precise timing information (on vehicle speed, frequency, etc.) provided to a traffic control algorithm spinning may cause system instability, which is why it is usually avoided in polling systems. These and other divergences between traffic control and CPU scheduling algorithms make the former difficult to apply directly to scheduling problems.

6.2 Scheduling practice

CPU scheduling algorithms used in practice can be roughly divided into those that are derived directly from theory and those that embody the systems approach, i.e. find out what works through educated guessing and trial-and-error and validate it after the fact.

6.2.1 Theory in practice

Until recently there was little crossover between the scheduling theory and systems communities. In published papers theoreticians would pay lip service to real-world applications before shifting to analytical results, while systems researchers focused largely on performance results, with little or no theoretical justification for the means of attaining them. In the last decade a number of leading authorities from the scheduling theory community have attempted to rectify this situation by applying a few simple analytical results to scheduling practice, in the hope of attracting system implementers with the lure of better performance.

These efforts were spearheaded by Harchol-Balter, whose papers on incorporating Shortest Remaining Processing Time-inspired heuristics into web servers [CFHB99, HBBSA01] have been heavily cited by the scheduling theory community. In [HBBSA01] Harchol-Balter et al. modified the Linux kernel to give priority to server threads that write less response data to the network over threads that write more data. In subsequent work Harchol-Balter and her collaborators investigated load balancing between web servers in a server farm [GHBSW07], limiting lock interference within databases by selectively admitting transactions [SWHB06], and scheduling in routers [BSUK07]. Harchol-Balter's research, particularly [HBBSA01], have been well received by the theory community, and have inspired third-party systems work in a similar vein [RK03], though the status quo in the systems community remains largely unaffected.

6.2.2 Systems approaches

Systems approaches to CPU scheduling typically focus on the robustness of implementations in actual use rather than the guarantees of theoretical models. The following sections briefly survey some alternative approaches to conventional priority-based thread schedulers (i.e. those descended from the BSD scheduler [MNN04]) that share some features in common with the present work.

Proportional share scheduling

The "excessive fairness" of the process schedulers found in many commodity operating systems has been the focus of extensive research over the last two decades. There have been many numerous attempts to develop schedulers that give developers more control over the kernel's scheduling decisions, though these proposals usually stop far short of abdicating control entirely (as in scheduler activations [ABLL91]). Notable among these efforts is lottery scheduling [WW94], which belongs to the class of proportional share schedulers. A proportional share scheduler divides available CPU time among processes according to fixed proportions (e.g. some percentage of available time), which are determined by some static [WW94] or dynamic [MP89, SGG+99] weighting scheme.

For high-performance servers the major advantage of proportional share scheduling in the kernel over conventional priority-based scheduling is that it allows servers to accurately and fairly divide CPU time among different threads/processes according to e.g. service classes. For this reason, among others, the default Linux kernel scheduler was recently redesigned as a proportional share scheduler, called the Completely Fair Scheduler [Mol07]. The default inputs to the scheduler are still `nice` values, past running and sleeping times, interactivity metrics, etc. rather than the proportions themselves, but instead of simply using these inputs as hints to decide which process to run at a switching point (as in BSD-derived schedulers), the inputs are translated into a target proportion and enforced over long periods.

In the absence of a proportional share scheduler in the kernel an application can approximate proportional share policies in userspace between the multiple threads or processes of the application [Reg01]. Although it was never explicitly presented as such, the SEDA technique of adding and removing kernel threads to/from a stage may also be viewed as a crude proportional share scheduler: adding a thread increases the amount of CPU time a stage receives over time, in effect increasing the stage's proportion. The technique is crude because it does not allow proportions smaller than a quantum, which is also not constant over time. From this perspective the thread-per-core stage scheduling policies come closer to real proportional share scheduling: over long periods of steady state execution the proportion of CPU time given to each stage should stabilize and remain almost constant. The MG1 algorithm makes these proportions explicit in its calculation of visit frequencies for each stage, though the actual observed percentage of CPU time spent processing events cannot be guaranteed on the scale of milliseconds or even seconds, since the scheduler is non-preemptive.

Adaptive real-time scheduling

CPU schedulers for hard and soft real-time systems [AB90] often share the basic mechanisms of kernel scheduling (processes, preemption, etc.) while diverging significantly in their scheduling policies. Real-time processes are usually associated with deadlines and target service rates, which transforms the scheduling problem into a constraint system.

Despite this fundamental difference, some recent work on so-called "real-rate" systems is of some interest to the present discussion, insofar as it indicates the extent to which additional information on process/stage CPU requirements could simplify scheduling algorithms and make them more predictable. In the late 1990s Steere and collaborators published a series of papers on real-rate schedulers, where "rate" refers to a target rate of a soft real-time process such as video streaming. With this target rate (e.g. the frame rate) in hand the CPU scheduler can dynamically adapt the proportion of CPU time given to each process in the application, giving a process more time on the CPU if it is not encoding frames fast enough to meet the target rate or deducting from a process's CPU proportion when the process has ample CPU time already. The general model is that of a producer (the video streaming process) and a consumer (a network user receiving frames) with a synchronized rate between the producer and consumer. The target rate may not always be specified explicitly as in the video streaming application: it might also be implicit in e.g. observing the level of a shared buffer or queue between the producer and consumer, in which case the scheduler would attempt to keep the buffer or queue half full by giving or taking CPU time to/from the producer or consumer.

Abeni et al. proposed a similar idea in their work on adaptive reservations in the Linux kernel [CPM+04]. Reservation-based scheduling is similar to proportional share scheduling in that it allows users and developers to assign a fraction of available CPU time to a process. Proportional share scheduling is slightly less conservative than reservation-based scheduling, however, in that the former only attempts to guarantee a given CPU proportion over relatively long periods of time and will compensate processes that do not use their entire quanta. Reservation-based scheduling more closely resembles a circuit-switched network, where every circuit is guaranteed a proportion of the available bandwidth even if this results in inefficient use of the available resources. In a reservation scheduling system short-term shortages (e.g. a dropped connection, missed scheduling deadlines) are only occasionally tolerated, unlike in a proportional share system, where it is assumed that processes can simply be compensated over time. Abeni's algorithm uses techniques from control theory to dynamically adjust process reservations in the kernel based on observed progress. It is based on the notion of a *virtual finishing time*, which is the deadline a process would finish at if it kept up its target rate. This is obviously quite similar to Steere's rate feedback, except that the constraints on deadlines/rates are tighter.

Other kernel modifications

While many researchers have recognized the importance of CPU scheduling to server performance, most research into improving CPU scheduling efficiency has focused on exploiting unmodified operating system kernels in unconventional ways or making minimally-invasive changes to these kernels (see section 2.1). However, a number of researchers have gone well beyond both points in pursuit of more extreme optimizations.

Bhatia et al. [BCL06] designed both a user-space memory manager and a kernel process scheduler specifically for event-driven servers. In order to identify regions of code that could benefit from increased L2 instruction and data cache re-use Bhatia introduced a limited notion of stages to describe the operations of a server. Here a stage is simply the code that processes events of a certain type(s) and produces events of another type(s); the concurrency model is still a single-threaded event loop. In Bhatia's architecture the memory allocator attempts to minimize the overlap between the working sets of two stages, while the scheduler (i.e. an event loop) receives feedback from the memory allocator regarding which objects are currently in cache. The authors incorporated their memory allocator and scheduling feedback mechanisms into several event-driven servers, including the TUX in-kernel web server. The resulting performance improvements are impressive, though it is unclear how generally applicable this approach is and how well it would work with servers that block frequently or are bound to a hardware resource other than the CPU.

As part of his PhD research in the mid-1990s Engler and his collaborators developed the Exokernel, "an operating system architecture that multiplexes machine resources while permitting an unprecedented degree of application-specific customization of traditional operating system abstractions" [EKO95]. The impetus behind the Exokernel is the same as that cited by Welsh in "Virtualization Considered Harmful" [WC01], namely that operating system abstractions such as processes and virtual memory that are designed to fairly multiplex underlying hardware resources severely hinder the performance of demanding applications such as servers. While Welsh's response to this problem was simply to repurpose the same abstractions, the Exokernel eliminated them entirely, reducing the kernel to its most basic hardware protection mechanisms. Application developers that needed higher-level abstractions such as network protocols or virtual memory could mix and match from a selection of application-level "modules" in which the same facilities were optimized for different purposes, much as developers of scientific computing applications can select from different linear algebra kernels. Less extreme and more practical versions of this approach were implemented in other operating systems, such as SPIN [BSP+95] and Nemesis [LMB+96], which allowed applications to heavily and directly influence scheduling and other kernel policies without the kernel abdicating its traditional functions.

6.3 Stage architectures

The principles of stage architectures have been applied to several software domains, from software routers to single node and distributed databases. Factoring code into event-passing stages offers numerous advantages over more traditional layered architectures, both in terms of design considerations (modularity, encapsulation) and runtime behavior (reconfigurability, data and instruction locality). The following sections survey the evolution of stage architectures for server-like systems from the late 1990s until the present.

JAWS

The advent of the World Wide Web in the early 1990s and subsequent research into the dynamics of highly concurrent web servers coincided with the then-current fascination with design patterns for software development. The JAWS web server framework was an embodiment of both trends [HS99], and one of the first widely-recognized attempts at building a high-performance web server. Unlike Flash and other highly-specialized servers, JAWS was a framework of frameworks that allowed developers to selectively re-implement parts of a server. It included frameworks for operating system I/O, protocol handling, concurrency/threading, and file caching [SH98]. Some of these frameworks may be viewed as stages, while others (such as the concurrency framework) approximate the function of SEDA's stage infrastructure.

The Click modular router

The Click modular router [KMC+00] is often cited as one of the first high-profile examples of a stage architecture like the ones described in this dissertation. Click was used to implement and benchmark routers for Ethernet, IP, BGP, DiffServ, and other protocols. The Click architecture allowed software developers to configure fine-grained packet processing elements in a directed graph. Unlike later event-driven architectures such as SEDA, Click packets could be pulled from one element to another or pushed from a producer to a consumer. Packets could also dynamically alter their path through the element graph based on information about the flow to which a packet belonged. A packet processing element in the Click architecture is not exactly equivalent to a stage, however, since some elements could be passive, such as e.g. queues that simply performed some operation whenever a packet was enqueued.

Click was initially targeted toward single processor commodity machines, and was only later adapted to SMP architectures [CM01b]. Scheduling in the Click router was straightforward compared to scheduling for servers: packet processing operations almost always have short and predictable execution times, which largely obviates the need for complex scheduling algorithms. The authors of [CM01b] did not attempt to optimize CPU scheduling on multiprocessors beyond basic load balancing between processors and allowing developers to statically assign packet processing elements to specific cores.

SEDA

The Staged Event Driven Architecture (SEDA) was originally designed as a refinement to the classic single-threaded event loop [WGBC00]. In SEDA the event loop was broken down into a series of stages separated by queues. The first SEDA technical report [WGBC00] describes several patterns for decomposing reactive event processing logic into pipelines of stages and replicating stages across multiple processors. Only later was the architecture associated with thread pool-per-stage concurrency ([WCB01] and section 2.4.1 of this dissertation).

After 2000, SEDA publications such as [WC03] and a culminating PhD thesis [Wel02] emphasized the "overload control" aspect of the architecture: by limiting the number of threads assigned to each stage, enforcing admission controls at each stage, allowing backpressure on queues, and selectively degrading service by e.g. decreasing the quality of images in a static file web server, the performance of a staged system could gracefully degrade under overload, staying at or near peak throughput. The resulting "load-conditioned services" were contrasted to thread-per-connection servers whose performance degraded severely when the servers were overloaded [WC01]. For his PhD thesis [Wel02]

Welsh benchmarked several SEDA-based servers, including a SPECweb99-like HTTP file server, a Gnutella router, and a webmail server with a database back end, in an effort to show that SEDA is well suited to many high-performance servers, in contrast to the HTTP-specific optimizations of Flash [PDZ99].

The main shortcoming of SEDA was its close association with thread pool-per-stage concurrency; increasing and decreasing the number of threads in a stage's thread pool was the primary mechanism of Welsh's per-stage controllers. In his thesis Welsh suggested that other concurrency strategies such as Cohort scheduling [LP02] could be substituted in order to e.g. emphasize cache re-use instead of overload control, but he did not pursue this course further.

StagedServer

The designers of the StagedServer architecture [LP02] emphasized a different aspect of stage architectures, namely increasing data and instruction cache re-use in staged servers through greater spatial and temporal locality. The StagedServer architecture was associated with a specific scheduling policy, Cohort scheduling (see section 2.4.2), which was designed to increase cache hit rates by batching similar computations (events). The StagedServer architecture also made several allowances for multiprocessor execution. Stages were classified as exclusive (e.g. single-threaded), partitioned (multithreaded but with per-processor data structures), and shared (multithreaded with global data structures). Events associated with the same request were processed on the same processor by default, though they could also be load-balanced between processors at programmer-defined junctures. The shared data structures of stages in a benchmark server were also partitioned across different processors in order to increase per-processor data cache locality.

Staged databases

A recent project at Carnegie Mellon [HA05] combined aspects of both SEDA and the Cohort scheduling/StagedServer paradigm in the design of a staged Database Management System (DBMS). Harizopoulos and his collaborators observed that the existing thread-per-connection concurrency of conventional DBMSs adversely affected the hardware efficiency of these servers, due in large part to the effects of thread context switching on both data and instruction cache utilization. The authors of [HA05] separated the operations of a monolithic DBMS (I/O, query parsing, optimization, execution of different relational operators) into a sequence of stages in order to increase instruction cache re-use by shortening code paths and increase the data cache hit rate by batch processing requests of the same type. Harizopoulos et al. chose a variation of the SEDA thread pool-per-stage scheduling policy for their architecture, with non-preemptive, user-level threads substituted for the the original SEDA's kernel threads (see section 2.4.2 for a discussion of this hybrid policy). The number of threads in a given stage's thread pool could be adapted according to client demand, similar to SEDA's controllers.

6.4 Programming language support for stage concurrency

Concurrent programming has long been a primary concern of the programming language research community. Most of the research in this area has focused on providing new high-level abstractions for concurrent programming to supplement or replace the current status quo of low-level threads and locks. These improvements are primarily targeted at mainstream procedural and object-oriented languages such as C or Java; in a functional language such as Haskell the type system can ensure that the segments of a threaded computation (between blocking operations) can safely be executed in parallel, in effect creating very fine-grained stages [LZ07].

Active objects

The favored contenders for these new procedural language abstractions are variants of active objects [SRSS00] and actors [Agh86], both of which bear some conceptual similarity to stages. In an active object system, method execution is decoupled from method invocation: rather than calling methods directly, code that invokes a method on an active object enqueues a request to the active object, which executes the method asynchronously. A scheduler within the active object decides which requests to execute next, and the results of a method execution are returned to the caller asynchronously. The model does not dictate a particular scheduling policy, though in practice active objects are often implemented with one user-level thread per object and a FCFS queue discipline, which has a clear analog in SEDA-like stages. An active object is not equivalent to a stage by the the definition of chapter 1, however, since there is no guarantee that methods on two active objects with different classes can run concurrently. In this respect an active object more closely resembles a monitor [Hoa74] than a stage.

Actors

Actors [Agh86] are similar to active objects in that they execute code asynchronously in reaction to messages. However, actors need not be objects with formal interfaces, but can dynamically route and process messages. As with active objects, different actors are not guaranteed to be concurrently executable, although like active objects, each actor is usually associated with a [user-level] thread, and sending a message from one actor to another crosses a thread boundary. Recent optimizations of actor-based systems have focused on reducing the overhead of message passing [FAH$^+$06, SM08] by eliminating copies and enforcing isolation between threads. Although scheduling the lightweight threads associated with actors has been less of a concern than correctness and other non-performance-related issues, the same techniques used to avoid copies between threads may also allow compilers to delimit stages [Sri07], which would make actor-based programs amenable to the scheduling optimizations described in this dissertation.

Both actors and active objects are general-purpose abstractions for concurrency that can be and have been adapted to mainstream programming languages such as C or Java without changing the languages themselves. An alternative approach is to extend these languages with new primitives for concurrent programming.

Cilk

Cilk is a C-based language and runtime for multithreaded programming [Blu95]. Cilk threads differ from the conventional kernel and user-level threads discussed in this dissertation in that they cannot block, but always run to completion. A thread can spawn off other threads, which return their results by invoking a continuation. This is similar to two stages communicating with events, although Cilk threads are not guaranteed to be concurrently executable. CPU scheduling optimizations for the Cilk runtime have centered on its work stealing thread scheduler [BL94]. Work stealing algorithms typically put precedence on balancing the load among processors in a multiprocessor machine ahead of other concerns, though later theoretical work on the Cilk algorithm included analysis of the data locality of work stealing [ABB00].

Flux

Flux is a domain-specific language for high-performance servers [BGK$^+$06]. A Flux program describes an acyclic graph for request processing logic, where nodes in the graph are C or C++ functions implementing some operation of the server (listening for connections, parsing HTTP requests, etc.) Given such a program, the Flux compiler can generate runnable server code with one of several different concurrency models, including thread-per-connection and single-threaded event-driven servers in C and SEDA servers in Java. The compiler also uses per-node mutual exclusion requirements specified by the programmer to check safety guarantees and to generate fine-grained reader/writer locking code for multithreaded runtimes. With sufficiently granular mutual exclusion constraints such an approach should be orthogonal to the explicitly-delimited stages considered here, and the server runtimes generated by the Flux compiler could likewise incorporate the stage scheduling policies investigated in this dissertation.

Identifying parallelism in event-driven servers

In the past proponents of the single-threaded event loop concurrently model have often boasted that the model eliminates the need for error-prone locking on shared resources by only relying on a single thread. With the advent of multicore processors the model's explicit limitation of parallelism has become a serious liability. A number of solutions to this problem have been proposed in the literature. These include simply replicating the single-threaded process multiple times [KKK07]; manually "coloring" event handlers to indicate concurrently-executable code [ZYD$^+$03]; and using compile-time static analysis to identify regions of safely-parallelizable code in existing event-driven servers [JP06]. The latter approach seems particularly promising as a means of reconciling legacy event-driven code with today's processor architectures, though it is unclear how useful this will be in the future, when programmers who set out to write new event-driven servers are more likely than their predecessors to consider multiprocessor issues from the outset.

Chapter 7

Conclusion

Stage architectures and their associated concurrency model have proved to be a superior alternative to conventional kernel thread-per-connection and single-threaded event-driven concurrency for high-performance servers. This proof was established by first generation research into stage architectures, which focused on comparing stage concurrency to other models in terms of overload handling behavior, data and instruction cache re-use, and raw throughput.

The research reported in this dissertation belongs to a second generation that builds on the basic premise that stages are a fundamentally superior way of handling concurrency compared to naive kernel thread-per-connection approaches. Rather than concentrating on the advantages of the staged concurrency model over other models, I have focused on variations within the same model, particularly scheduling policies for staged servers. The main positive results of this research came from a series of experiments that demonstrated that (1) thread-per-core policies outperform thread pool-per-stage policies in terms of throughput on realistic CPU-intensive workloads and (2) explicit load balancing can improve the throughput of thread-per-core-scheduled servers. The same experiments also produced a negative result, namely that the thread-per-core policies as a class are competitive with each other on current hardware. More generally, these results ratified the known performance advantages of staged concurrency over other concurrency models and confirmed that stage architectures are a worthwhile subject for future research.

7.1 Future work

Inter-stage queues

With only a few cores per machine the queues and stacks of current stage architectures are usually not a major factor in staged server performance, because the cost of passing events between stages is far outweighed by the cost of processing the events. The number of cores in commodity machines is set to rise dramatically in the future, however, and this will increase the relevance of inter-stage communications mechanisms. In particular, I expect to see non-blocking data structures replace the current standard of locked, blocked queues and stacks in userspace servers.

In addition to the benefits of non-blocking operation itself, there are a number of optimizations that decrease the cost of inter-stage communication via these mechanisms. For example, by making some assumptions about the number of producer stages or the number of consumer stages – usually assuming the number of producers or consumers is one – some signaling between producers and consumers can be avoided [Hup02]. More generally, non-blocking data structures can be optimized for specific use cases by e.g. attaching the list of consumers waiting for events to the sentinel element of the non-blocking queue rather than having a separate semaphore for the wait list, as in the current event queue implementation in Yield. For stages that can tolerate event reordering, substituting an efficient non-blocking stack [Boe04] for a current non-blocking queue could also increase the data cache hit rate of a server. For applications such as the image processing server a lock-free priority queue [ST05] could also impose an SRPT queue discipline on individual stages, so that smaller images would always have precedent and could clear the system quickly.

Models

In the last decade a range of models and heuristics from queuing theory to control engineering have been applied to web server admission control and scheduling, yet no approach has emerged as demonstratively more accurate and effective in comparison to the others. Although many of these techniques were inherited from the study of network congestion, the variable cost of web server connections and requests (vs. the constant cost of routing packets), the more complex nature of web server code (vs. relatively simple routing algorithms), and the general unpredictability of the web environment (vs. the statistical characterization of networks) has made web server performance hard to pin down. Similarities between staged servers and queueing networks can simplify performance analysis for these servers in particular, though relatively few researchers have attempted to do so [Wel02, LLCZ06].

Benchmarks

There is a real need for more rigorous benchmarks for HTTP, SMTP, and other servers that isolate the performance of the server independently of the larger system while still being realistic. Designing and implementing rigorous independent benchmarks is a notoriously difficult task with very few tangible rewards, which discourages many qualified and unbiased participants. Nevertheless, this is a necessary condition for future research in high-performance servers, which must move beyond the status quo of HTTP file server benchmarks.

Optimizing CPU scheduling for other targets

Rising energy costs and environmental concerns have spurred many companies and researchers to consider power consumption as a fundamental design parameter of computing systems. Modern processors decrease power consumption in hardware by switching to low power states when peak performance is not required. Moreover, running one of today's multiprocessor machines at peak performance can actually slow them down: the cooling devices in many commodity machines are not able to dissipate enough heat over prolonged periods of time, so the processor(s) must decrease their clock speeds preventatively in order to avoid overheating.

The majority of software CPU schedulers, in contrast, are designed to raise throughput or meet response times even if this also induces unnecessary power consumption. Recently a number of researchers have reconsidered these basic goals of CPU scheduling. To cite one example, Merkel and his collaborator have designed and implemented a Linux kernel process scheduler that selects processes to run and/or migrates them between cores with the explicit goal of reducing power consumption [MB08]. The scheduler accomplishes this by tracking and observing the way processes access the functional units of a processor and ensuring that no unit is used excessively. For instance, a process that uses the floating point units much more than the integer units might be scheduled between two integer-heavy processes, so that the integer units can cool down while the floating point-heavy process is running. Allowing the units to cool down means that the clock speed of a given core can remain high without causing overheating, which usually means that more instructions can be executed. This in turn implies that fewer cores are required to handle the same load. Cores that are not needed can run in low power states, so less power is consumed.

Unfortunately, the fine-grained, system-wide observation and control in Merkel's scheduler is difficult to replicate accurately outside the kernel. A user-level scheduler such as those described in this dissertation would have to rely on user input to indicate which functional units were being accessed, and even then there would be no guarantee that some other process outside the user-level scheduler's control (but within the kernel's) was not perturbing the system. A more promising approach would be to run the server on a single core by default, and have a strategy for including and decommissioning more cores as the need arose, as indicated by some Quality of Service metric such as response times.

7.2 Acknowledgments

David Eyers, a post-doctoral researcher at the University of Cambridge Computer Laboratory, assisted me in setting up and running the RUBiS benchmark by installing and configuring the RUBiS MySQL database and benchmarking the Java servlets implementation of RUBiS.

Thomas Lange, proprietor of `wie-ich.de`, provided the data set for the image processing server benchmark and the original batch script upon which the server was based.

Oiestein Andersen, Ted Briscoe, and the Natural Language Processing Group at the Computer Laboratory kindly provided the source to the Robust Accurate Statistical Parsing (RASP) library for use in the site search benchmark.

The system administrators at the Computer Laboratory and at the Darwin Computer resolved many technical issues I encountered in the last three years. Darwin is run by the University of Cambridge High Performance Computing Service and was provided by Dell Inc. using Strategic Research Infrastructure Funding from the Higher Education Funding Council for England.

I would like to personally thank my supervisor, Jean Bacon, for making things easier when I insisted on making them difficult; David Eyers, for his patience, persistence, and shell scripting fu; Sriram Srinivasan, for sharing his perspective; David Evans, for not taking me too seriously; Felix Hupfeld, for showing me much more than the debugger; and my family, for reminding me where I came from and why I'm here.

Appendix A

Appendix: implementation notes

A.1 Cohort scheduling: per-core stacks

In addition to SEDA-like synchronized event queues, stages in a StagedServer-based application also incorporated per-core event stacks as a means of increasing data cache re-use. From [LP02]:

> A stage maintains a stack and queue for each [core] in the system. In general, [events] originating on the local [core] are pushed on the stack and [events] from other [cores] are enqueued on the queue. When a stage starts processing a cohort [of events], it first empties its stack in LIFO order, before turning to the queue. This scheme has two rationales. Processing the [most recent events] first increases the likelihood that an [event's] data will reside in the cache. In addition, the stack does not require synchronization, since it is only accessed by one [core], which reduces the common-case cost of [sending an event].

Unfortunately, from the above description it is not entirely clear when an event for a given stage should be put on a core's stack at that stage and when it should be directed to the stage's single event queue. Consider a system with two processors, P1 and P2, and three stages, S1, S2, and S3. When P1 is visiting S1 and P2 is visiting S2, and S1 generates an event for S2, where does the event go?

1. To P1's per-core stack on S2? If that is the case, then there is very little use for the queues.
2. To the queue on S2, on the (worst-case) assumption that S2 will be visited by an "other processor" before P1 can visit it (and process its stack)? If this worst-case assumption holds then the stacks are not very useful, since they could only hold events that were generated for S1 by S1. If P1 happened to be processing events from S1's queue when the event for S1 was generated, then the latter event would not be processed until P1 visited S1 again (since stacks are visited before queues), which would lessen its chances of staying in cache. Furthermore, in real servers stages rarely send events to themselves, so the stacks would go unused most of the time.

For the experiments in chapter 4 I implemented per-core stacks for Cohort scheduling according to interpretation 1, where events for a stage always go to its per-core stacks, except when

1. an event is enqueued by some outside thread that is not part of the thread-per-core set (such as a set of threads for offloading blocking disk operations); or
2. a stage is single-threaded and must be locked. Upon encountering a single-threaded stage a thread/core does not block waiting for the lock, but moves on to the next stage if the lock cannot be acquired immediately. This means that the inter-visit times for a particular stage/core combination can be very long, which in turn implies that an event on a per-core stack could wait a long time to be processed.

In both cases the stage's synchronized queue is the more appropriate destination for an event.

Note that the use of a per-core stack implies the possibility of event reordering. This is not acceptable for many applications, which do not account for arbitrary event arrival orders in single threaded code, such as, for example, buffers for an HTTP request body arriving before the HTTP headers. For this reason as well as the unclear delineation of queues and stacks my use of per-core stacks was limited to benchmarking Cohort scheduling.

Bibliography

[AB90] N. Audsley and A. Burns. Real-time System Scheduling. Technical Report YCS 134, University of York, 1990.

[ABB00] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Theory of Computing Systems*, pages 1–12. ACM Press, 2000.

[ABC$^+$05] Eric Anderson, Dirk Beyer, Kamalika Chaudhuri, Terence Kelly, Norman Salazar, Cipriano Santos, Ram Swaminathan, Robert Tarjan, Janet Wiener, and Yunhong Zhou. Value-maximizing deadline scheduling and its application to animation rendering. In *SPAA '05: Proceedings of the Seventeeth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 299–308, New York, NY, USA, 2005. ACM.

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *SOSP '91: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, New York, NY, USA, 1991. ACM.

[ADCM98] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring. Scheduling with implicit information in distributed systems. *SIGMETRICS Performance Evaluation Review*, 26(1):233–243, 1998.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

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

[ALL89] T. E. Anderson, D. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. In *SIGMETRICS '89: Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 49–60, New York, NY, USA, 1989. ACM.

[Amd67] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, pages 483–485, 1967.

[App92] Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, New York, NY, USA, 1992.

[Arm97] Joe Armstrong. The development of Erlang. *SIGPLAN Not.*, 32(8):196–203, 1997.

[BC01] Laura Bertolotti and Mariacarla Calzarossa. Models of mail server workloads. *Perform. Eval.*, 46(2-3):65–76, 2001.

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* O'Reilly & Associates Inc, 2005.

[BCL06] Sapan Bhatia, Charles Consel, and Julia Lawall. Memory-manager/scheduler co-design: optimizing event-driven servers to improve cache behavior. In *Proceedings of the 2006 International Symposium on Memory Management (ISSM'06)*, pages 104–114, Ottawa, Canada, June 2006. ACM.

[BCW06] Ted Briscoe, John Carroll, and Rebecca Watson. The second release of the rasp system. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, pages 77–80, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

[BD03] Jason Brittain and Ian F. Darwin. *Tomcat: the definitive guide.* O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, 2003.

[BDM98] Gaurav Bangs, Peter Druschel, and Jeffrey C. Mogul. Better operating system features for faster network servers. *SIGMETRICS Performance Evaluation Review*, 26(3):23–30, 1998.

[Ber95] Daniel J. Bernstein. Using maildir format. `http://cr.yp.to/proto/maildir.html`, 1995.

[BGK+06] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: a language for programming high-performance servers. In *ATEC '06: Proceedings of the USENIX Annual Conference*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.

[BL94] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[Blu95] Robert Blumofe. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[BLW93] Onno J. Boxma, Hanoch Levy, and Jan A. Weststrate. Efficient visit orders for polling systems. *Performance Evaluation*, 18(2):103–123, 1993.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Notices*, 35(11):117–128, 2000.

[Boe04] Hans-J. Boehm. An almost non-blocking stack. In *PODC '04: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pages 40–49, New York, NY, USA, 2004. ACM.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Notices*, 40(6):261–268, 2005.

[BP05] James R. Bulpin and Ian A. Pratt. Hyper-threading aware process scheduling heuristics. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 27–27, Berkeley, CA, USA, 2005. USENIX Association.

[BPG04] Tim Brecht, David Pariag, and Louay Gammo. accept()able strategies for improving web server performance. In *USENIX Annual Technical Conference, General Track*, pages 227–240, 2004.

[BSP$^+$95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[BSUK07] Ernst W. Biersack, Bianca Schroeder, and Guillaume Urvoy-Keller. Scheduling in practice. *SIGMETRICS Performance Evaluation Review*, 34(4):21–28, 2007.

[CCE$^+$03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Middleware'03: Proceedings of the 4th Middleware Conference*, 2003.

[CCN$^+$05] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 36–36, Berkeley, CA, USA, 2005. USENIX Association.

[CDN$^+$96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *ATEC '96: Proceedings of the Annual USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 1996. USENIX Association.

[CFHB99] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[CGK$^+$07] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA '07: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[Cla04] Richard Clayton. Stopping spam by extrusion detection. In *CEAS 2004 - First Conference on Email and Anti-Spam*, July 2004.

[Cla05] Richard Clayton. Stopping outgoing spam by examining incoming server logs. In *CEAS 2005 - Second Conference on Email and Anti-Spam*, July 2005.

[CM01a] Abhishek Chandra and David Mosberger. Scalability of linux event-dispatch mechanisms. In *Proceedings of the General Track: 2001 USENIX Annual Technical*, pages 231–244. USENIX, 2001.

[CM01b] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor PC router. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, pages 333–346, Boston, Massachusetts, June 2001.

[CPM$^+$04] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a Linux environment. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 238, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[CSO99] Feng Cheng, Suresh P. Sethi, and Tava Lennon Olsen. A practical scheduling method for multiclass production systems with setups. *Management Science*, 45(1):116–130, 1999.

[DB99] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. In *SOSP '99: Proceedings of the Seventeeth ACM Symposium on Operating Systems Principles*, pages 247–260, New York, NY, USA, 1999. ACM.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. *SIGOPS Operating Systems Review*, 25(5):122–136, 1991.

[DL05] J. G. Dai and Wuqin Lin. Maximum pressure policies in stochastic processing networks. *Operations Research*, 53(2):197–218, 2005.

[DN03] Vincent Danjean and Raymond Namyst. Controlling kernel scheduling from user space: An approach to enhancing applications' reactivity to I/O events. In *HiPC*, pages 490–499, 2003.

[DSPPR$^+$86] Jr. D S Parker, G J Popek, G Rudisin, A Stoughton, B J Walker, E Walton, J M Chow, D Edwards, S Kiser, and C Kline. *Detection of mutual inconsistency in distributed systems*, pages 306–312. Artech House, Inc., Norwood, MA, USA, 1986.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[Duf08] Joe Duffy. *Concurrent Programming on Windows.* Addison-Wesley Professional, Boston, MA, USA, November 2008.

[Dus07] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[Eng00] Ralf S. Engelschall. Portable multithreading: the signal stack trick for user-space thread creation. In *ATEC '00: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 177–190, New York, NY, USA, 2006. ACM.

[Fet05] Abe Fettig. *Twisted Network Programming Essentials.* O'Reilly Media, Inc., 2005.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[FJL+01] Françoise Fabret, H. Arno Jacobsen, François Llirbat, Joăo Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 115–126, New York, NY, USA, 2001. ACM.

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: language support for event-driven programming. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 134–143, New York, NY, USA, 2007. ACM.

[FRK02] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the Ottawa Linux Symposium 2002*, June 2002.

[FSSN05] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *ATEC '05: Proceedings of USENIX Annual Technical Conference*, pages 26–26, Berkeley, CA, USA, 2005. USENIX Association.

[GCA+07] Luiz Henrique Gomes, Cristiano Cazita, Jussara M. Almeida, Virgílio Almeida, and Jr. Wagner Meira. Workload models of spam and legitimate e-mails. *Performance Evaluation*, 64(7-8):690–714, 2007.

[GHBSW07] Varun Gupta, Mor Harchol-Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9-12):1062–1081, 2007.

[Gol90] E.M. Goldratt. *Theory of Constraints.* North River Press, Croton-on-Hudson, N.Y., 1990.

[Gor05] Minor Gordon. Staged design for highly concurrent web servers. Master's thesis, Technical University of Berlin, 2005.

[GR04] D. Gilgen and T. Rowe. United states patent 20040199926: Enhanced staged event-driven architecture, 2004.

[HA02] Stavros Harizopoulos and Anastassia Ailamaki. Affinity scheduling in staged server architectures. Technical report, Carnegie Mellon University, 2002.

[HA05] Stavros Harizopoulos and Anastassia Ailamaki. Stageddb: Designing database servers for modern hardware. *IEEE Data Engineering Bulletin*, 28(2):11–16, 2005.

[HBBSA01] Mor Harchol-Balter, Nikhil Bansal, Bianca Schroeder, and Mukesh Agrawal. SRPT scheduling for web servers. In *JSSPP*, pages 11–20, 2001.

[HD94] H-U. Heiss and M. Dormanns. Mapping tasks to processors with the aid of Kohonen-networks. In *Proceedings of the High Performance Computing Conference (HPCC)*, September 1994.

[Hel05] D. Helbing. *Production, Supply, and Traffic Systems: A Unified Description*, pages 173–188. Springer Berlin Heidelberg, 2005.

[HG06] Felix Hupfeld and Minor Gordon. Using distributed consistent branching for efficient reconciliation of mobile workspaces. In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (IEEE CollaborateCom) Workshops*, pages 1–9, 2006.

[HGLS86] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[HPJ$^+$07] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on Chip Multiprocessors: Limitations and Opportunities. In *Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, pages 79–87, 2007.

[HS99] James C. Hu and Douglas C. Schmidt. Jaws: A framework for high performance web servers. In *Domain-Specific Application Frameworks: Frameworks Experience by Industry.* Wiley & Sons, 1999.

[Hup02] Felix Hupfeld. Design and performance of a user-level network driver in a multi-server operating system. Master's thesis, University of Karlsruhe, 2002.

[IEE88] IEEE. *1003.1-1988 INT/1992 Edition, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988)*. IEEE, New York, NY, USA, 1988.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–329, New York, NY, USA, 1988. ACM.

[JP06] John Jannotti and Kiran Pamnany. Safe at any speed: fast, safe parallelism in servers. In *HOTDEP'06: Proceedings of the 2nd Conference on Hot Topics in System Dependability*, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[JR98] A. Jones and J. Rabelo. Survey of job shop scheduling techniques. Technical report, NISTIR, National Institute of Standards and Technology, 1998.

[JV06] Tatuya Jinmei and Paul Vixie. Implementation and evaluation of moderate parallelism in the BIND9 DNS server. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[KC07] Evangelia Kalyvianaki and Themistoklis Charalambous. On dynamic resource provisioning for consolidated servers in virtualized data centers. In *Proceedings of the 8th International Workshop on Performability Modeling of Computer and Communication Systems, (PMCCS-8)*, 2007.

[KKK07] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *ATC'07: 2007 Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(1):291–307, 1970.

[KMC$^+$00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[KS03] Christopher Kohlhoff and Robert Steele. Evaluating soap for high performance business applications: Real-time trading systems. In *WWW'03: the Twelfth International World Wide Web Conference*, May 2003.

[Lee06] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006.

[LH08] Stefan Lämmer and Dirk Helbing. Self-control of traffic lights and vehicle flows in urban road networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(04):P04019 (34pp), 2008.

[LL02] Ben Laurie and Peter Laurie. *Apache: The Definitive Guide.* O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, third edition, 2002.

[LLCZ06] Zhanwen Li, David Levy, Shiping Chen, and John Zic. Auto-tune design and evaluation on staged event-driven architecture. In *MODDM '06: Proceedings of the 1st Workshop on MOdel Driven Development for Middleware (MODDM '06)*, pages 1–6, New York, NY, USA, 2006. ACM.

[LMB$^+$96] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

[LN79] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

[LP02] James R. Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *ATEC '02: Proceedings of the 2002 USENIX Annual Technical Conference*, pages 103–114, Berkeley, CA, USA, 2002. USENIX Association.

[Lur06] Marty Lurie. Websphere tuning for the impatient: How to get 80% of the performance improvement with 20% of the effort. *IBM developerWorks web site*, February 2006.

[LZ07] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 189–199, June 2007.

[Mau99] Jim Mauro. The Solaris process model: Managing thread execution and wait times in the system clock handler. *SunWorld*, March 1999.

[MB08] Andreas Merkel and Frank Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 1–12, 2008.

[MEG03] Luke K. McDowell, Susan J. Eggers, and Steven D. Gribble. Improving server software support for simultaneous multithreaded processors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, New York, NY, USA, 2003. ACM.

[MJ98] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[MNN04] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System.* Pearson Education, 2004.

[Moc87] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987.

[Mol07] Ingo Molnar. Linux: Discussing the completely fair scheduler. `http://kerneltrap.org/node/8208`, 2007.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[MRYGM01] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. In *WWW '01: Proceedings of the 10th International Conference on World Wide Web*, pages 396–406, New York, NY, USA, 2001. ACM.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, New York, NY, USA, 1996. ACM.

[Nah02] E. Nahum. Deconstructing SPECweb99. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, August 2002.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[NP07] T. Newhouse and J. Pasquale. Achieving efficiency and accuracy in the alps application-level proportional-share scheduler. *Journal of Grid Computing*, 5(2):251–270, June 2007.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes), January 1996.

[PBH$^+$07] David Pariag, Tim Brecht, Ashif S. Harji, Peter A. Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. In *EuroSys*, pages 231–243, 2007.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[Phi00] Lawrence Philips. The double metaphone search algorithm. *C/C++ Users Journal*, 18(6):38–43, 2000.

[Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, 3(14):130–137, October 1980.

[PRH07] A. Pathak, S. Roy, and Y. Hu. A case for a spam-aware mail server architecture. In *CEAS 2007 - Fourth Conference on Email and Anti-Spam*, August 2007.

[RBH06] Andrew C. Rice, Alastair R. Beresford, and Robert K. Harle. Cantag: an open source software toolkit for designing and deploying marker-based vision systems. In *PERCOM '06: Proceedings of the Fourth Annual IEEE*

International Conference on Pervasive Computing and Communications, pages 12–21, Washington, DC, USA, 2006. IEEE Computer Society.

[RCL01] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The Oracle database resource manager: Scheduling CPU resources at the application level. In *Proceedings of the 9th International Workshop on High Performance Transaction Systems (HPTS 2001)*, 2001.

[Reg01] John Regehr. Some guidelines for proportional share CPU scheduling in general-purpose operating systems. In *Work in Progress session of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London, UK, 2001.

[RK03] Mayank Rawat and Ajay Kshemkalyani. SWIFT: scheduling in web servers for fast response time. In *NCA '03: Proceedings of the Second IEEE International Symposium on Network Computing and Applications*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[RLA07] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. Thread scheduling for multi-core platforms. In *HOTOS'07: Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[RP04] Yaoping Ruan and Vivek S. Pai. The origins of network server latency & the myth of connection scheduling. In *SIGMETRICS*, pages 424–425, 2004.

[SACY04] Christian Siefkes, Fidelis Assis, Shalendra Chhabra, and William S. Yerazunis. Combining winnow and orthogonal sparse bigrams for incremental spam filtering. In *PKDD '04: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 410–421, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[SGG$^+$99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 145–158, Berkeley, CA, USA, 1999. USENIX Association.

[SH98] Douglas C. Schmidt and James C. Hu. Developing flexible and high-performance web servers with frameworks and patterns. *ACM Computing Surveys*, page 39, 1998.

[SKK00] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference*, pages 296–310, August 2000.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*. Springer, 2008.

[SMD07] Fengguang Song, Shirley Moore, and Jack Dongarra. Feedback-directed thread scheduling with memory considerations. In *HPDC '07: Proceedings of the 16th International Symposium on High Performance Distributed Computing*, pages 97–106, New York, NY, USA, 2007. ACM.

[SPM07] S. Siddha, V. Pallipadi, and A. Mallick. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal*, 11(4), 11 2007.

[Sri07] Sriram Srinivasan. Personal Correspondence, 2007.

[SRSS00] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

[ST05] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computation*, 65(5):609–627, 2005.

[ST07] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007.* SIAM, January 2007.

[Sta96] Standard Performance Evaluation Corporation. An explanation of the SPECweb96 benchmark. `http://www.spec.org/osg/web96/webpaper.html`, 1996.

[Sta99] Standard Performance Evaluation Corporation. SPECweb99 release 1.02. `http://www.spec.org/web99/docs/whitepaper.html`, 1999.

[Sta00] Standard Performance Evaluation Corporation. SPEC CPU2000 v1.3 documentation. `http://www.spec.org/cpu/docs/`, 2000.

[Sta01] Standard Performance Evaluation Corporation. Specmail2001 mail server benchmark architecture white paper. `http://www.spec.org/mail2001/docs/whitepaper.html`, 2001.

[SWHB06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd Conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[SZ92] Mark L. Spearman and Michael A. Zazanis. Push and pull production systems: issues and comparisons. *Operations Research*, 40(3):521–532, 1992.

[TAS07] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, New York, NY, USA, 2007. ACM.

[TTG93] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary. *SIGMETRICS Performance Evaluation Review*, 21(1):272–274, 1993.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th Conference on Hot Topics in Operating Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[vBCZ$^+$03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281, New York, NY, USA, 2003. ACM.

[Wal91] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[WC01] Matt Welsh and David E. Culler. Virtualization considered harmful: Os design directions for well-conditioned services. In *HotOS*, pages 139–144, 2001.

[WC03] Matt Welsh and David E. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[Wel02] Matthew David Welsh. *An architecture for highly concurrent, well-conditioned internet services*. PhD thesis, University of California, Berkeley, 2002.

[WGBC00] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, EECS Department, University of California, Berkeley, 2000.

[Wil02] Nathan J. Williams. An implementation of scheduler activations on the netbsd operating system. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 99–108, Berkeley, CA, USA, 2002. USENIX Association.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, pages 1–11, 1994.

[WWB07] Adam Wierman, Erik M. M. Winands, and Onno J. Boxma. Scheduling in polling systems. *Performance Evaluation*, 64(9-12):1009–1028, 2007.

[YRP$^+$07] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 93–104, New York, NY, USA, 2007. ACM.

[ZYD$^+$03] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.