Improving the Performance of User-level Runtime Systems for Concurrent Applications

by

Saman Barghi

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

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Carsten Griwodz Professor, Department of Informatics, University of Oslo, Oslo, Norway Chief Research Scientist, Simula Research Laboratory, Lysaker, Norway
Supervisor(s):	Martin Karsten Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal Member:	Peter A. Buhr Associate Professor, Cheriton School of Computer Science, University of Waterloo
	Ken Salem Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Werner Dietl Assistant Professor, Dept. of Electrical and Computer Engineering, University of Waterloo

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

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

Abstract

Concurrency is an essential part of many modern large-scale software systems. Applications must handle millions of simultaneous requests from millions of connected devices. Handling such a large number of concurrent requests requires runtime systems that efficiently manage concurrency and communication among tasks in an application across multiple cores. Existing low-level programming techniques provide scalable solutions with low overhead, but require non-linear control flow. Alternative approaches to concurrent programming, such as Erlang and Go, support linear control flow by mapping multiple user-level execution entities across multiple kernel threads (M:N threading). However, these systems provide comprehensive execution environments that make it difficult to assess the performance impact of user-level runtimes in isolation.

This thesis presents a nimble M:N user-level threading runtime that closes this conceptual gap and provides a software infrastructure to precisely study the performance impact of user-level threading. Multiple design alternatives are presented and evaluated for scheduling, I/O multiplexing, and synchronization components of the runtime. The performance of the runtime is evaluated in comparison to event-driven software, system-level threading, and other user-level threading runtimes. An experimental evaluation is conducted using benchmark programs, as well as the popular Memcached application. The user-level runtime supports high levels of concurrency without sacrificing application performance. In addition, the user-level scheduling problem is studied in the context of an existing actor runtime that maps multiple actors to multiple kernel-level threads. In particular, two locality-aware work-stealing schedulers are proposed and evaluated. It is shown that locality-aware scheduling can significantly improve the performance of a class of applications with a high level of concurrency. In general, the performance and resource utilization of large-scale concurrent applications depends on the level of concurrency that can be expressed by the programming model. This fundamental effect is studied by refining and customizing existing concurrency models.

Acknowledgements

I would like to take this opportunity to thank all those who helped me to make this thesis possible. First and foremost, I would like to express my sincere thanks to my supervisor, Martin Karsten, for his regular advice, guidance, encouragement and perseverance throughout the course of my PhD. This thesis is only made possible with his unlimited help and support.

I would also like to thank Peter A. Buhr for his guidance, help, and support throughout the course of my PhD.

I thank the members of my PhD committee for their valuable and constructive feedback: Carsten Griwodz, Weinter Dietl, Ken Salem, and Peter A. Buhr.

Finally, I would like to thank the many friends and colleagues I have met during my studies for their support and friendship over the past few years.

Dedication

This is dedicated to my beloved parents and family for their love, endless support, encouragement, and sacrifices.

Table of Contents

List of Tables

List of Figures

Abbreviations

AIO The POSIX Asynchronous I/O 19, 22, 33

API Application Programming Interface 9, 14, 23, 33, 35, 40, 62

CAF C++ Actor Framework 32, 33, 93, 94, 99–102, 107, 120

CMP Chip Multicore Processor 5, 6

CPU Central Processing Unit 5, 7, 8, 13, 14, 18, 20, 21, 25, 26, 30, 39, 46, 48, 72, 83, 86, 121, 145

CSP Communicating Sequential Processes 17, 19, 34, 36, 37, 92, 145

CV Coefficient of Variation 63, 116

DAG Directed Acyclic Graph 25, 30

DNS Domain Name Server 15

ET Edge Triggered 24, 36, 37

FD File Descriptor 24, 41, 43, 52–58, 67, 71

FIFO First In First Out 27, 29, 33, 36, 43, 86, 94, 95

GDB GNU Project Debugger 34

GHC Glasgow Haskell Compiler 37

HCS Hierarchical Chunk Selection 95

HPC High Performance Computing 25, 30–32, 38, 39, 46, 121, 144

HTTP Hypertext Transfer Protocol 64, 66, 67, 76

HVS Hierarchical Victim Selection 95

I/O Input/Output vii, viii, x, xi, 2, 3, 6, 9, 12–15, 18–26, 30–33, 35–37, 39–43, 51, 52, 54, 55, 63, 64, 66, 69, 70, 75, 80, 90, 119, 121, 145

IPC Instruction Per Cycle 11

LAS Locality-Aware Scheduler ix, 97, 100

LAS/A Locality-Aware Scheduler/Affinity 100–107

LAS/L Locality-Aware Scheduler/Local 100–107, 109

LIFO Last In First Out 29, 36, 94, 95

LT Level Triggered 32

MPI Message Passing Interface 145

MPI Message Passing Interface 144

MPSC Multiple Producer Single Consumer 28, 62, 66, 95

MVA Mean Value Analysis 110, 115

NIC Network Interface Controller 144

NUMA Non-Uniform Memory Access 6, 7, 31, 40, 41, 62, 93, 95–100, 102, 103, 105, 106, 120, 145

OS Operating System 8, 10, 11, 52, 53

PGAS Partitioned Global Address Space 145

POSIX Portable Operating System Interface 33, 34

RWS Randomized Work Stealing 30, 92, 93, 101

SEDA Staged Event-Driven Architecture 18, 33

SMP Symmetric Multi-Processors 5

TLB Translation Look-aside Buffer 11

UMA Uniform Memory Access 6

Chapter 1

Introduction

Concurrency is an essential part of many modern large-scale software systems with the rise of many-core and distributed platforms. Applications must handle millions of simultaneous requests from millions of connected devices. Each request is comprised of one or multiple operations that often lead to creating more concurrency. Efficient handling of such a large number of concurrent requests demands proper distribution of tasks among cores of a machine. Such requirements encourage the use of programming models and runtime systems that efficiently manage concurrency and communication among tasks in an application. Event-based, thread-based, and actor programming are among the main programming models that support large-scale concurrent applications.

As systems evolve from handling tens of thousands of connections [107] to handling tens of millions of connections [81] on a single machine, creating high-performance concurrent runtime systems becomes crucial. On the other hand, due to the recent breakdown of Dennard scaling [31] modern computer chips utilize multi-core processors, rather than increasing frequency, to improve performance. Therefore, further application performance improvement has to come from exploiting parallelism by using multiple threads of execution. Hence, modern runtime systems need to be scalable across many cores to take advantage of modern computer chips. Programming models provide the semantics for producing concurrent software, but the performance and scalability of concurrent programs are determined by the runtime system.

The formulation of the C10K problem [107], which addresses the inability of servers to scale beyond ten thousand clients at the time, has encouraged the adoption of event-based programming in the absence of scalable user-level threads [213]. Fully event-based runtime systems are single threaded and exploit non-blocking I/O operations to efficiently serve tens

of thousands of connections on a single core. These systems only add a thin layer on top of the kernel and have very low overhead for handling I/O-bound workloads. However, event programming requires explicit capturing of state and has a non-linear control flow that adds to the complexity of developing applications. Furthermore, any blocking operation, such as blocking I/O operations, can block the single thread of execution and pause the application. To address this problem, simple hybrid thread/event solutions use delegation to execute blocking operations in background threads [133, 161, 206].

Modern servers are expected to handle ten million simultaneous connections (C10M [81]), which is only possible through utilizing multiple cores. This in turn requires exploiting multiple threads of execution. Since fully event-based systems are single-threaded and, even twenty years after the formulation of the C10K problem, there is still no highly scalable and low-overhead thread runtime, event-based solutions are combined with threading to support large-scale concurrency. A simple variant (N-copy) does not support shared mutable state, while more complex hybrid solutions that share memory among threads require a mix of event and thread programming. These approaches do not provide proper load-balancing or fine-grained control over the executing threads. Most importantly, hybrid thread/event systems must address the complexities of both event and thread programming at the same time. Thus, developing applications becomes even more difficult.

These shortcomings provide the motivation to build more efficient runtime systems that have support for concurrency and implicit state capture for simpler application development. Such systems should also provide flexible load-management, transparent scalability, and support concurrent execution of both CPU-bound and I/O-bound tasks over many cores. Kernel-level runtime systems have relatively high overhead in handling concurrency, since the operating system must provide fairness and protection at the kernel level. On the other hand, user-level runtime systems have lower overhead with the assumption that scheduling tasks within a single application does not require protection and fairness at the application level. Thus, switching among user-level tasks is relatively cheaper because switching between user-level entities that store state does not involve the kernel. Fine-grained multi-threading, where multiple user-level threads are mapped to multiple kernel threads, provides a promising alternative for high performance and scalable runtime systems with implicit state capture.

1.1 Thesis Statement

The main thesis of this work is showing that it is possible to build an efficient, low-overhead, and scalable user-level runtime system for handling large-scale concurrency. The user-level

runtime presented in this thesis is a nimble cooperative user-threading library providing a proving ground for testing and evaluating different algorithms and data structures for scalable multi-core scheduling and I/O demultiplexing. This runtime provides a blocking threading abstraction with extremely low runtime-overhead and has been compared to the fastest runtime systems currently on the market [201]. The library is written in C++ but strives to be a bare-bones threading-library to prevent conflating language features with pure runtime-performance. The goal is to precisely understand the cause and effect of performance changes to gain true insight into runtime alternatives and their benefit with respect to specific workloads.

Chapter 2 provides some background on user-level runtimes and studies existing user-level threading runtimes. The central thesis is substantiated and investigated using a software prototype for a user-level threading runtime described system in Chapter 3. The investigation is broken down into three components that form the typical bottlenecks of a user-level runtime: scheduling, synchronization, and I/O handling. The runtime and its different components are evaluated in Chapter 4. As an extension of this work, a locality-aware work stealing scheduler for user-level runtimes based on the actor model is proposed and evaluated in Chapter 5. The performance and resource utilization of large-scale concurrent applications depends on the level of concurrency that can be expressed by the programming model. This fundamental effect is studied by refining and customizing existing concurrency models in Chapter 6.

1.2 Main Results

Building a low-overhead user-level runtime requires design, implementation, and optimization of different components of the runtime. This thesis studies different design alternatives for each component and the key findings and contributions are listed below.

A simple multi-queue scheduler with work-stealing for a user-level threading runtime is presented, which is sufficient to achieve good resource utilization. This scheduler is evaluated across different scenarios and is shown to have minimal overhead for the, sometimes very short, user-level thread invocations. In addition, two locality-aware work-stealing schedulers are proposed and evaluated for actor runtimes. It is shown that a locality-aware scheduler significantly improves the performance of actor-based applications for some workloads.

A flat and a hierarchical design alternative are presented for I/O polling and it is shown that offloading blocking polling to a separate system thread allows for a very simple im-

plementation with good performance and effective thread-idle management. Both designs require no configuration and are shown to have good performance across different scenarios.

This thesis also explores synchronization primitives that block at the user-level. While this subject has been studied in the literature, this thesis confirms that any mutex design with barging is largely sufficient for real applications. In summary, the thesis delivers an existence proof for low-overhead user-level concurrency by demonstrating performance that is unprecedented in the literature for openly available software.

Chapter 2

Background and Related Work

2.1 Multicore Processors

Modern computers utilize multicore processors to increase computing performance, since the breakdown of Dennard scaling [31] makes substantial increases of clock frequencies unlikely and because the depth and complexity of instruction pipelines have also reached a breaking point [14]. Among these architectures, Chip Multicore Processor (CMP) has emerged as the dominant architecture choice for modern general-purpose computing platforms. CMP architecture introduces a new set of challenges both for software programmers and hardware designers. Using many cores on the same chip rather than on a single high-frequency Central Processing Unit (CPU) means software programmers can not benefit from serial performance improvements anymore [192]. Future increases in performance need to come from parallel computing through exploiting multiple threads of execution. This transition means that programming languages, compilers, operating systems, and software development tools need to evolve to adopt the new trend of hardware design. These topics have been addressed in the 1990s by the advent of Symmetric Multi-Processors (SMP), but CMP and new architecture designs introduce new challenges.

The hardware architecture substantially affects parallel application performance [93], and therefore various chip designs have been developed to address the needs of different applications. These designs differ in internal circuitry of processors, number of processors, arrangement of processors and memory modules in an architecture, the communication mechanism among the processors and between the processors and memory modules, number of instruction and data streams, nature of memory connections with the processors, nature and types of interconnection among the processors, and program overlapping [207].

CMP architectures are classified into two major categories based on their memory arrangement:

- Shared-memory multiprocessor: Multiple processors share a common memory unit consists of single or multiple memory modules. The communication among processors is established through writing to and reading from the shared memory. Both sequential applications and parallel applications can be executed over the shared memory architecture. However, concurrent access to the same memory location should be coordinated through a proper parallel or concurrent programming model which is discussed further in Section 2.2. In addition, when all processors try to access the global memory at the same time, this creates a bottleneck that limits the scalability of the system. To address this problem large shared memory systems have some form of hierarchical or distributed memory structure.

- Distributed-memory multiprocessor: This architecture consists of multiple units that each is a complete computer building block including a processor, memory and I/O systems. Each processor can only access its local memory unit and communication among processors is established through direct I/O operations. Although direct processor-to-processor communication adds overhead but this design does not suffer from the same bottleneck problem as shared-memory architectures. However, as the number of processors increases, it becomes impractical to directly connect all processors to each other.

The shared-memory multiprocessor is the most commonly used architecture among multiprocessors. Shared memory designs are grouped in two levels of symmetry. In Uniform Memory Access (UMA) architectures, all processing units share the same main memory and I/O facilities. These processors are connected by a bus-based interconnection such that the memory access time is approximately the same for all the processor units. However, hardware scaling limitations have led to the emergence of Non-Uniform Memory Access (NUMA) multi-chip platforms [65] as a trade-off between low-latency and symmetric memory access.

Contemporary large-scale shared-memory computer systems are built using NUMA hardware where each processor chip has direct access to a part of the overall system memory, while access to the remaining remote memory (which is local to other processor chips) is routed through a slower interconnect [141]. A NUMA architecture introduces additional challenges for programmers to avoid high latency due to remote NUMA node memory access and at the same time avert increasing contention over the resources on the local chip [129, 26].

The difference between CPU frequency and the frequency of the memory bus causes the performance of main memory to lag behind the CPU performance. One solution to narrow the gap between memory and CPU is to have a small amount of high-speed cache memory in addition to a large amount of slower main memory. The faster layer acts as a cache for the slow main memory and a temporary store for the data that is used by the processor. As the speed difference between cache and main memory increases, there is a need to add another level of slower and bigger cache between the first level and main memory. Today, there are systems that have up to four levels of cache.

CPU caches help to improve the performance because of *locality of references*: both spatial and temporal. Spatial locality indicates data or code residing in close storage locations is referenced in a short period of time. Temporal Locality indicates the same code or data is reused in a short time period. Temporal locality is a special case of the spatial locality, in other words, when the prospective location is identical to the present location.

Depending on the cache level, the difference between memory and cache access latency can be between 4 to 10 times [169, 194]. This difference shows that the performance penalty of cache misses at various levels is very high, and programmers should try to improve cache locality and increase the cache hit-rate in programs to avoid these penalties and improve the overall performance.

In addition, when a core accesses a memory location on a remote NUMA node, it requires communication across one or multiple interconnects. The associated cost of such communication is called the "NUMA factor". The NUMA factor represents the ratio between remote latency and local latency among NUMA nodes. The NUMA factor depends on the number of interconnects that the core has to communicate across to reach the target NUMA node. If all nodes are directly connected to each other, the cost of accessing all remote nodes from cores on a CPU is the same. Otherwise, the cost of accessing different NUMA nodes varies.

Although NUMA improves scalability, the negative effect of resource contention is still significant [227, 26]. Another issue is related to increased overhead of inter-processor thread migrations– i.e., a thread allocates memory on one node and migrates to another processor attached to a different NUMA node where accessing the original memory has higher latency [26, 123]. To avoid inter-process thread migrations it is possible to set processor affinity for each thread [149]. However, this limits the scheduler options and in some cases it can further hurt the performance. Therefore, effective scheduling algorithms are required to avoid such performance issues [109].

Clearly, with remote memory, the efficiency of the CPU caching hierarchy becomes

even more important. However, CPU caches are shared among cores with the particular nature of sharing depending on the particulars of the hardware architecture. Moreover, applications that employ user-level scheduling add to the complexity of the problem, as the Operating System (OS) does not know about the details of tasks and kernel-level scheduling loses its efficiency. Therefore, new scheduling algorithms are proposed that are aware of the memory hierarchy and shared resources to exploit cache and memory locality and minimize overhead [224, 84, 140, 72, 147, 228, 188].

2.2 Concurrent and Parallel Programming Models

Parallel programming is used to exploit multiple processing elements within a single machine. Parallel programming is substantially more difficult than sequential programming due to the added complexity of the *coordination* aspects [153, 193, 175], i.e., how the computation is decomposed, mapped to, communicated among, and synchronized across the available processing elements.

Concurrency is used in many contexts to provide responsiveness, fault tolerance, exploit parallelism, and making software more efficient. Concurrency is widely used in network servers, high performance computing, databases, big data, and many more areas. One of the popular applications of concurrency is network servers where an application has to handle many concurrent connections. Such applications exploit single or multiple threads of execution to receive, process, and respond to concurrent requests coming from many clients. Although all aspects of concurrent software are discussed in this thesis, the focus is placed on the network applications that handle many concurrent requests. Hence, the models presented here are mainly studied from the point of view of such applications.

Concurrent programming models determine the type of task abstraction, method of synchronization and communication among tasks, and method of state capture. Each model uses a different form of abstraction to represent tasks, such as, threads, events, and actors. Each model determines how tasks can communicate and coordinate using either the shared-memory model or message-passing model. Tasks communicate through accessing some shared memory location. If a memory location is not shared and is not being accessed by other tasks, there is no need for coordination. Furthermore, if the memory location is shared but is immutable, i.e., the state cannot be modified after it is created, there is no need for coordinating accesses among tasks. Therefore, tasks only have to coordinate access to shared mutable memory locations to guarantee correctness.

When a task is blocked, its state must be captured to be restored later when it is unblocked. Depending on the model, state capture is either explicit and performed by the

application, or implicit and is performed by the runtime through a form of continuation, such as a task's private stack.

Each model provides either a synchronous or an asynchronous programming interface. With a synchronous programming interface, code is executed sequentially from top to bottom, except for conditionals and function calls. Blocking or busy waiting is used to wait for external events, such as the network and disk I/O, to keep the order of execution sequential. For instance, each thread in thread-programming provides a synchronous programming interface.

Alternatively, with asynchronous programming, part of the code is not executed sequentially. In order to wait for an external event, the program sends a request to the external source and keeps processing other work. The program detects when the result of the request is ready by receiving an interrupt or through periodical polling. State capture is done explicitly through a *callback function* that is registered when the requests is sent. When the result of the request is ready, the *callback function* is used to restore state and handle the result.

Asynchronous programming with explicit state capture is considered to be hard, as raising requests from within another request can cause heavy chaining of callbacks, which is referred to as "delayed GOTO" [86]. Asynchronous programming creates error-prone and difficult to extend code that is hard to read and follow. Furthermore, if state transfer to another part of the code is necessary, state must be explicitly saved and resumed later, which is known as *stack ripping* [6]. Some languages provide closures or continuations to address this issue, but it remains a problem for low-level languages such as C. Event-based programming provides an asynchronous programming interface and requires explicit state capture by the application.

For instance, using a synchronous model of programming for interacting with nanosecond- and millisecond-scale devices is claimed to be easier than the asynchronous model [23]. It is argued that synchronous code is a lot simpler, and therefore easier to write, tune, and debug. The benefits are further amplified at scale where synchronous programming provides a simple and consistent Application Programming Interface (API) across various programming languages and therefore improves development productivity. In addition, synchronous programming takes away the burden of managing asynchronous requests from the programmer to the operating system or runtime and makes the code significantly simpler.

Ultimately, if performance is not an issue, programming models that support a synchronous programming interface and implicit state capture are more desired. Unlike event-based systems, a thread programming model satisfies these requirements. However, there has been no serious recent effort to study the performance of fine-grained multi-threading

for network servers on modern multi-core hardware. Hence, one of the objectives of this thesis is to identify performance bottlenecks and build a low-overhead and high performance threading library for handling large concurrency. Thread programming, event programming, and the actor model are now discussed in detail.

2.2.1 Thread Programming

Originally, threads were introduced to replace processes in UNIX systems to provide multi-path execution without the expense of inter-process communication to express concurrency [135]. Creating multiple processes adds substantial overhead due to the large amount of resources that each process requires. Furthermore, UNIX systems do not provide an appropriate way of sharing resources such as network connections among processes. Therefore, the thread abstraction was introduced as a single computational unit that runs within a process address space and shares resources with other threads in the same address space.

Threads are the natural extension of the dominant sequential programming style for providing concurrency in most programming languages [213, 214, 86]. Threads allow the developer to focus on sequential steps of the operation while developing concurrent code. Each thread has its own stack where it manages its state. Threads provide a simple and powerful abstraction when tasks are mostly isolated and only share a limited amount of state.

Threads can communicate by using the shared memory or message passing model. The shared memory model is the most common model used in thread programming. The most well-known drawback of using threads with the shared memory model [150] is the difficulty of developing correct concurrent code. As soon as there is state sharing among multiple threads, coordination and synchronization become imperative. Sharing requires usage of mutual-exclusion locks, which can cause deadlocks. Choosing the right lock granularity is another problem; coarse locks slow down concurrent code and too fine locks increase the danger of deadlocks. Moreover, given two thread-safe components, combining them does not always guarantee a new thread-safe component.

Threads are used to realize concurrency, as an organizational tool, and to exploit multiple processors. The OS kernel provides *system concurrency* through kernel-level threads and multiplexes them over multiple processors. These threads can be used to realize concurrency or provide parallelism. User-level thread libraries provide *user concurrency* through user-level threads, which are not recognized by the kernel, and require scheduling and management in user space. These threads cannot be used to support parallelism, but provide a more natural programming model for concurrency.

Kernel threads are scheduled by the operating system, usually using a preemptive scheduler. Hence, when a kernel thread does a context switch it needs to issue a system call. Furthermore, switching the context of kernel-level threads happens via a system call. Each system call requires switching from user-space to kernel-space, perform the context switch, and switch back to user-space. Therefore the context switch is relatively expensive and causes flushing of the user-space pipeline, saving a few registers onto the kernel stack, and changing the protection domain. The total round-trip time for entering and leaving the kernel space is estimated to be 150 cycles [177]. In addition, the context switch over head is shown to be noticeable [218]. Creating and joining an OS-level thread involves significant overhead, especially when the number of threads is more than the number of physical cores.

In addition, there are other indirect performance costs associated with system calls. These costs arise from the system-call footprint where processor structures including L1 data and instruction caches, Translation Look-aside Buffer (TLB), branch prediction tables, prefetch buffers, as well as larger unified caches are populated with kernel specific state. These direct and indirect overhead sources affect user Instruction Per Cycle (IPC) [177].

Although, kernel-level threads are considered to have large overhead, it is shown in Chapter 4, if the kernel is tuned properly, the latest version of pthreads in Linux can be used to handle 500,000 connections simultaneously with relatively high throughput. Creating and destroying threads are very expensive, but as long as a thread-pool is used to serve connections, performance is not affected. In addition, since these workloads rely on many system calls to interact with the network, these tasks are relatively coarse grained so the overhead of system calls used for context switches does not affect the performance significantly as this cost does not dominate the work.

However, for more fine grained tasks the context switch overhead becomes significant. Moreover, the lack of control over the OS scheduler and details of thread placement might prevent the application from reaching its scheduling objectives. For instance, handling a large number of connections using pthreads leads to high variations in the latency, which is not desired for a network application. Also, kernel threads also consume a large amount of resources, and creating an application with a very large number of kernel threads can slow down the application and the whole system.

To address these problems, user-level threads[1] are exploited to provide higher levels of concurrency. Context switches in user-level threads happen in user-space and therefore

[1]User-level threads are also called, green threads (Java), fibres (Windows), and state threads (state thread library) [173]

avoid the complexity and overhead of kernel-level context switches. In addition, this provides the opportunity to associate user-level threads with fine grained tasks that translates to smaller stack space and lower memory footprint. In addition, since the scheduler is in user-space, it provides fine-grained control over scheduling and mapping user-level threads to processors.

User-level threads are mapped to kernel-level threads to be executed with different policies: 1:1, N:1, and M:N. 1:1 mapping means one user-level context is mapped to one kernel-level thread. This mapping is the default in many systems, and there is no duality between user-level and kernel-level threads.

With N:1 mapping, multiple user-level threads are mapped to a single kernel-level thread. There is no need for load-balancing as there is only a single kernel-level thread that is executing the user-level threads. This mapping can only keep a single processor busy and cannot be used to exploit multiple cores. Scheduling is usually performed cooperatively among user-level threads, where each user-level thread cooperatively yields to another user-level thread, or blocks on an event, e.g, network I/O. Since blocking the underlying kernel thread blocks all user-level threads, blocking operations must not block the kernel-level. For instance, network I/O operations must not block the underlying kernel thread. Instead of blocking the kernel thread, user-level threads are blocked waiting for I/O using non-blocking operations and the runtime transfers the control to another user-level thread.

In order to exploit multiple processors, multiple copies of a process or thread with N:1 mapping can be used, and this variant is called N-copy. However, due to the lack of central scheduling, a user-level thread cannot move to another kernel thread after it is assigned to a kernel thread. Hence, load-balancing can become an issue.

M:N mapping multiplexes multiple user-level threads over multiple kernel-level threads. Kernel-level threads are used to realize parallelism and user-level threads provide additional concurrency. However, to properly utilize all cores, user-level threads should be distributed among kernel-level threads. This distribution requires load-balancing and scheduling techniques, which are investigated later in this chapter.

Since the kernel does not have any knowledge about user-level threads, user-level threads face several problems. For instance, if a kernel thread that is executing a user-level thread holding a spin lock is preempted by the kernel, other user-level threads might end up spinning for a long time. In addition, the kernel might preempt a kernel thread that runs a high-priority user-level thread and replace it by another kernel thread that runs a lower-priority user-level threads.

Consequently, the *scheduler activations* [11] was proposed to address these problems. Scheduler activations provided kernel support for user-level management of parallelism. A

Scheduler activation is an execution context similar to a kernel thread that can be used to execute user-level threads. The kernel makes an *upcall* to inform the user-level threading library about the events that happened to a scheduler activation, e.g., the scheduler activation is preempted. The user-level library can make a decision regarding running the current user-level thread and other user-level threads that were scheduled to run on the activation. However, this design suffers from scalability and complexity problems and requires support from the operating system [189].

2.2.2 Event Programming

Event programming is a paradigm where events (e.g., network, keyboard, or mouse events) trigger the execution of related tasks. Tasks are started when an event happens and finish running when the task is done. Tasks can trigger new events (therefore triggering the execution of new tasks) while they are running. Each event is linked to a callback function that executes the intended task. Event-based applications use a single-threaded event loop that listens for events and triggers the callback function when an event happens. Event programming provides an asynchronous programming interface.

Events along with their callback functions represent a task in event-based systems. Using a single-threaded event loop eases concurrency concerns, and is regarded as a appropriate foundation for high performance servers [150]. Although most event-based applications are single-threaded, technically, if the event queue is thread-safe it can be shared among multiple threads, but this remove the main advantage of event-based applications which is ease of concurrency concerns. By using non-blocking I/O, multiple I/O operations overlap, thus I/O concurrency does not require CPU concurrency. Callbacks and event handlers can be used without coordination and synchronization. The execution of callbacks is guaranteed to be deterministic, as long as no yielding operation is triggered in the callback. The application is more aware of the scheduling, and developers can fine-tune the scheduling based on their application requirements. The event-driven design uses an asynchronous behaviour, which makes the differences between I/O operations and CPU-bound operations obvious.

Moreover, event-based programming is considered to have a low memory footprint. Events coupled with callback functions and explicit capturing of state, means task memory footprints can be reduced to the bare minimum. For low-level languages that do not support closures (such as C), it is developer's responsibility to explicitly save and restore states. This responsibility puts an additional burden on the programmer, and makes it hard to write and debug event-based applications. The control flow of an event-driven program

is not sequential since the function calls are intertwined and hard to follow, which makes debugging hard. Conversely, other programming models that rely on implicit capturing of state usually allocate extra memory space for each task. This additional space might not be used, e.g., extra stack space in the thread programming model.

In addition, single-threaded event-driven applications are vulnerable to long running CPU-bound callbacks, blocking operations, or callbacks that are not yielding. These callbacks can stall the application and increase latency. Even when these applications are using I/O concurrency, there is no simple solution to take advantage of CPU concurrency and utilize multiple processors for CPU-bound workloads.

On the other hand, the N-copy variant is widely used to exploit multiple processors. This approach relies on creating multiple copies of the event-based application, hence, creating multiple copy of the same process. Each process runs a single-threaded event loop and occupies a core. However, N-copy does not provide shared mutable state among processes, so there is no load balancing or fine-grained control over the execution of threads. Moreover, if multiple threads are used instead of multiple processes, then shared mutable state requires synchronization primitives, which has the same difficulties as stated in the thread programming model.

Event-driven network applications are built on asynchronous non-blocking events based on `select`, `poll`, and `epoll` call semantics. With this model, multiple connections are multiplexed over a single thread where connections are handled using non-blocking I/O semantics.

Non-sequential execution can make developing event-based applications very challenging. A solution to this problem is provided by *libasync-smp* [226], that tries to take advantage of multiple processors and asynchronous programming by assigning "colours" to different callbacks. Callbacks with the same col or must be executed sequentially, but callbacks with different colours can be executed concurrently. This approach requires reasoning about the execution flow and labelling callbacks correctly. Event-driven programming by itself is already difficult due to a complex control flow, and adding additional complexity can complicate it further. Furthermore, there are other solutions to address this problem using a state machine that avoids the complexities of callbacks, but these solutions cannot fully solve the problem.

There are several popular frameworks and applications based on event-based programming. *libevent* [133] replaces the event loop found in event-driven network servers and provides asynchronous event notification. It currently supports */dev/poll, kqueue(), event ports, POSIX select(), Windows select(), poll(), and epoll()* mechanisms, and exposes an API that is independent of the underlying event mechanism. Libevent hides the complex-

ities of the underlying I/O operations. Libevent is designed to be single threaded, but applications such as Memcached uses an event loop per thread (N-copy) and multiplex the connections over different threads.

libev [116] is a POSIX only event library and loosely modelled after libevent, but without its limitations. libev provides timers and better support for event loops to be used in a multi-threaded applications by avoiding global variables and only use per loop contexts. However, it does not provide some of the features of libevent such as support for HTTP and Domain Name Server (DNS) servers, but it can be combined with libeio [115] to implement those features.

libuv [2] was mainly developed to be used by Node.js [206] by polishing and removing some of the unnecessary functionalities in libev. Although originally based on libev, libev is no longer used. Libuv provides support for asynchronous I/O by using a single main thread that serves connections and a set of background threads to serve file I/O requests. Node.js is a popular event-based framework for developing web applications. It has been widely used among developers to provide lightweight application servers to serve thousands of requests per second with limited resources.

Nginx [161] is a high performance event-based load-balancer, web server, and reverse proxy. Nginx does not provide multi-threading support but it can be configured to run multiple copies of the same process.

ULib [183] is an event-driven web server that is among the top performer in TechEmpower benchmarks [201]. These benchmarks are designed to stress test different frameworks for creating web server applications. ULib uses an N-copy model to exploit parallelism, by pre-forking multiple processes where each use an event loop to serve connections. ULib is written in C++ and provides a modular design to add and remove routes based on pre-compiled shared libraries.

Memcached [62] implements a high-performance, distributed key-value memory object caching system. Since threads use shared memory, access to shared objects must be synchronized among threads. Therefore, Memcached relies on synchronization primitives for such coordination and uses mutex and condition variables from the *pthreads* library to protect objects and perform cleanups.

2.2.3 Actor-Model Programming

The actor model is a general-purpose programming model that provides fault tolerance and resilience, and facilitates geographical distribution. The actor model uses the message-passing memory model and can be used with both shared-memory and distributed-memory

architectures. Since actors do not share mutable state, there is no need for explicit synchronization among actors. This makes actor programs scalable both horizontally and vertically and provides location transparency. Similar to packet switch networks, programmers do not need to deal with the routing of messages and only need to provide the address of the recipient actor. The runtime is responsible for locating and routing the message to the recipient actor.

The actor programming model uses the term *actor* to describe autonomous objects that communicate asynchronously through messages. Each actor has a unique address that is used to send messages to that actor, and each actor has a mailbox that is used to queue and process received messages. Actors do not share state and only communicate by sending messages to each other. Sending a message is a non-blocking operation and an actor processes each message in a single atomic step.

Actors may perform three types of action as a result of receiving a message: *(1) send messages to other actors, (2) create new actors, (3) update their local state* [7, 8]. Actors can change their behaviour as a result of updating their local state. In principle, message processing in a system of actors is non-deterministic, because reliable delivery and arrival order of messages are not guaranteed. This non-deterministic nature of actors makes it hard to predict their behaviour based on static compile-time analysis or dynamic analysis at runtime.

In addition, some implementations of the actor model, such as, Erlang [212] and AKKA [32] provide fault tolerance concurrency. Fault tolerance is only possible through a managed language such as Erlang or Java where failed actor can be isolated and restarted. The fault tolerance in such environments is oppose to exception and error handling and provides an additional safety net to protect the application from bad behaviour of a single actor.

De Koster et al. [58] classify actor languages into four major paradigms: Classic Actors, Active Objects, Processes, and Communicating Event-Loops. Akka [32] is an example for a classic actor model based on Agha's work [7, 8]. An active object has a single entry point that defines a fixed interface of messages that are understood. Changes to the state of an actor can be done imperatively and isolation is guaranteed by sending passive objects between active objects by copy. Orleans [25] and SALSA [211] implement the active object model. Erlang [12] models actors as processes that run from start to completion. Processes have a single entry point with a flexible interface that can change by evaluating different receive expressions over time. Communicating event-loops defines a type of actor with multiple points of entry by using a combination of object heap, event queue, and an event loop.

The actor model is gaining popularity due to the increasing need for scalable and distributed applications. However, developing actor-based applications can be confusing and hard for programmers. For instance, Tasharofi et al. [197] demonstrates that many programmers use other concurrency models (such as, locks and condition variables) along with message passing to address concurrency issues. The problem results from weaknesses in actor library implementations and shortcomings of the actor model itself.

In addition, the size of messages and the additional internal synchronization required to send messages can affect performance and latency of applications. Messages are copied so they are never shared state. Hence, there is no reason to make the messages immutable from a concurrency perspective, unless the message contains pointers to shared state. An actor system can only work if messages are logically copied.

Moreover, there are other problems with the actor model that makes it suitable only for a specific range of applications [127]. The actor model has no direct notion of inheritance or hierarchy, which makes it confusing to develop applications that require this structure. The decision of where to store and execute newly created actors is important to overall performance. Unbounded mailboxes require some form of infinite stack or pointer structure which is not available on all systems. Since messages do not arrive in order, it may cause problems under certain circumstances (e.g., manipulating stack-like structures). If an actor requires a reply from another actor, it must stop and wait for the reply to keep atomic processing of messages in tact and therefore it cannot process other messages. The implementation details to support this feature varies among different frameworks, some frameworks block the underlying kernel-level thread and others block the user-level actor.

2.2.4 Other Programming Models

There are other concurrency programming models that are not the main subject of this thesis. However, since some of these models are interesting, a brief description of each model is presented.

Communicating Sequential Processes (CSP) [97] is a formal language for concurrent systems that is based on message passing via channels. CSP is widely used as a tool for specifying and verifying the concurrent aspects of systems. However, it also influenced the design of some programming languages such as Occam [41], Golang [79], and RaftLib [24].

CSP is different from actor model because CSP processes are anonymous, while actor processes have identities. Moreover, in CSP, message passing is synchronous so the sender process cannot transmit the message until the receiver is ready to accept it. However, message passing in actor systems is asynchronous so it is entirely decoupled between a

sender and receiver. This difference makes actors much more independent and therefore more suitable to run in a distributed environment.

Functional programming, in contrast to imperative programming, models computation as the evaluation of mathematical functions instead of a sequence of statements that change state when executed. Mathematical functions in functional programming are side-effect free, i.e., these functions only add new state in a local context. In other words, operations in a function does not modify the state of contexts outside of that function. Functional programming languages are usually less efficient than imperative languages when it comes to CPU and memory usage. Functional programs are mostly used in academia but not widely used in industry. However, functional languages such as Erlang, that combine the actor model with functional programming, are gaining popularity.

Moreover, there are other approaches that combine thread-based and event-based programming models to avoid their flaws and benefit from their advantages. Staged Event-Driven Architecture (SEDA) [220] is a design for highly concurrent internet services. It is an attempt to support massive concurrency demands by combining thread and event-driven programming models. The authors argue that thread programming is expensive due to context switch overhead and memory footprint, and cannot support concurrency when the loads are high. They also argue that event-driven programming is very challenging for the application developer, and it is hard to schedule and order events. SEDA, on the other hand, breaks the program into different stages, where each stage has its own thread pool, event queue, event handler and controllers.

SEDA employs various queueing techniques by handling the load on event queues at each stage. SEDA also facilitates debugging and performance analysis of services. Each stage comes with a set of resource controller tools, which automatically adapt the resource usage of that stage based on observed performance and demand: a thread-pool controller and a batching controller. The thread pool controller controls the number of threads in each stage, and the batching controller adjusts the number of events processed by each invocation of the event handler within a stage. SEDA is based on *Sandstorm*, and implemented in JAVA, and it uses non-blocking socket I/O.

The SEDA author later states [219] that having multiple stages causes multiple context switches by passing through various thread pools, and potentially long queueing at busy stages. These problems can lead to poor cache behaviour and increases the response time. One possible solution is to assign multiple stages to the same thread pool, and only context switch when there is I/O or non-deterministic runtime. Capriccio [214] was an effort to port SEDA to C, where a user-level thread library is used to avoid extensive context switches by using cooperative scheduling.

Li and Zdancewic [121] combine events and threads for massive concurrent network applications using monads in Haskell. Their hybrid model takes advantage of an event-driven system to handle asynchronous I/O using non-blocking interfaces, while using thread pools to perform synchronous, blocking operations. Unlike pure user-level threading systems, the scheduler is not hidden from the programmer, providing an event-driven model to interact with asynchronous I/O interfaces. Haskell monads are used to provide an abstraction over continuation passing and thread interfaces, so programmers interact with a synchronous thread-like interface. Multiple threads can be executed with each running an event loop, and scheduling of events is delegated to the user. This framework supports epoll and The POSIX Asynchronous I/O (AIO) to interact with network interfaces and process disk accesses in the background.

2.3 User-level Runtime Systems

Although programming models determine the characteristics of concurrent applications, the efficiency and performance of an application is highly dependent on the implementation of the runtime system. The runtime is responsible for efficient multiplexing of tasks (user-level entities) over kernel-level threads and assigning kernel threads to processors. Also, the runtime is in charge of properly balancing the load among kernel threads for efficient resource utilization and improving cache locality. In addition, the runtime interacts with I/O devices, such as, the network and file system. The runtime also provides synchronization primitives or proper tools to enforce the requirements defined by the programming model, e.g., mailboxes for actor model, channels for CSP, and mutex locks and condition variables for thread programming.

Finally, the runtime must provide I/O event demultiplexer and scheduling, which are both discussed in the following subsections. The runtime is responsible for making sure the application meets its various objectives such as low latency, high throughput, fairness, and efficient utilization of resources.

In addition, the user-level runtime provides both low level and high level services to help the programmers reach their objectives. The low level services include interfacing processors and peripherals, memory loading, synchronization primitives, etc. Higher level functions include debugging, code generation, and optimization services. This thesis is studying the low level services of user-level runtime systems, such as interacting with I/O devices and scheduling tasks across multiple cores.

2.3.1 I/O

Since the focus of this thesis is large-scale high-performance server applications, dealing with I/O overhead in the user-level runtime becomes an important factor. Such applications are I/O bound and reading from and writing to sockets and files are performed frequently. Hence, masking I/O delay is necessary to achieve good performance in a large-scale network server.

In this section, user-space interfaces to interact with I/O are classified by three different models and each model is studied in relation to user-level runtime systems. The models presented here are analyzed based on Linux, but the same concepts apply to FreeBSD. The system call interface for both Linux and FreeBSD is reviewed.

I/O interfaces are classified based on whether the interface is synchronous, asynchronous, blocking, or non-blocking. Completing an I/O operation synchronously means the entire path for handling an I/O request stays withing the context of the task that initiated the I/O operation. On the other hand, asynchronous I/O means part of the path to check the readiness of file descriptors is delegated to another thread or task. Blocking I/O means the task or thread running the I/O operation is blocked immediately or sometime in the future waiting for I/O. In contrast, non-blocking I/O means no user thread or task is blocked waiting for I/O.

Hence, there are three models based on the criteria above: *(1) Synchronous Blocking, (2) Synchronous Non-blocking, and (3) Asynchronous*. These models are discussed respectively in the following. It is important to note that these are high level models describing application I/O as a whole, and are different from individual I/O system calls. For example, even when *read* system calls are non-blocking, the application I/O model is considered blocking if the application has to block on polling for events.

I/O Interfaces

Synchronous Blocking I/O

In this model, a user program issues a system call to perform I/O, which causes a context switch to kernel space and the application blocks until the action is completed by the kernel. Blocking means the application (thread) does not consume CPU cycles, and another thread can take over the processor. When the kernel receives the system call, it initiates I/O activity on behalf of the user program. As soon as the response arrives from the device, the kernel moves the data to a user-space buffer and the application unblocks.

This model is the most common and well understood model. It works well with multi-threaded applications where I/O can be multiplexed among kernel-level threads. Hence, while one thread blocks on a system call, other threads take over the CPU and run. For example, the Apache web server [71] uses synchronous blocking I/O by default.

In addition, this model provides a nice sequential programming interface to programmers. Programmers do not need to worry about when the result of the I/O call is ready and masking I/O delay is done automatically by the kernel through switching the context to another thread.

Synchronous blocking can be used in the user-level, where a user-level thread is blocked on I/O and the runtime is responsible for polling the I/O devices and notifying the user-level thread when an event is triggered. The user-level runtime interface provides a wrapper around the system call and the programmer does not directly issue a system call for I/O operations.

Synchronous Non-Blocking I/O

In this model, instead of blocking and waiting for the I/O to be completed, the I/O system call returns immediately. For instance, if an application is interacting with network I/O through connections, if the I/O operation cannot be completed, an error code is returned. Thus, the application has to poll the kernel until the connection is ready.

This model is less efficient than synchronous blocking model since polling the kernel requires repeating the system call, and in many cases the application must busy-wait until the connection becomes ready. Busy-waiting translates to wasting CPU cycles and other processes or threads cannot use the CPU. Moreover, if the application does not busy-wait and attempts to do other work while waiting for the connection to become ready, it can introduce additional I/O latency or lead to starvation. For instance, if the system is under heavy load, polling for the readiness of a connection might never happen and the connection starves.

Furthermore, with this method, the programmer has to be aware of the underlying mechanisms, and needs to keep track of failed operations and ensure each I/O operation is polled frequently enough to avoid long latency. Also, when it comes to network I/O, this method is only efficient with active connections. When there are idle connections the overhead of unnecessary polling can increase significantly and the extra work required to keep track of all idle connections by the programmer renders this approach unusable. Due to the above problems, this approach is not being used widely in large-scale server applications.

Essentially, when this model is used along with busy-waiting, the application is blocking

the execution of the current kernel-level thread until the result is ready. Therefore, busy-waiting turns this model to a less efficient version of the synchronous blocking model.

Asynchronous I/O

This model uses non-blocking system calls for I/O operations, but a task can delegate the polling of devices to another task or thread. Multiple tasks pass multiple file descriptors to the kernel and one or more tasks poll the status of all file descriptors at once. This approach is more efficient than polling each file descriptor individually.

Moreover, this model makes it possible to multiplex multiple file descriptors over a single kernel-level thread. Since I/O operations are non-blocking, if the file descriptor is not ready, the thread switches to another file descriptor or performs other computations. Polling is batched and is done by a thread asynchronously.

In Linux, this model can be implemented by using `select, poll,` or `epoll` system calls along with non-blocking I/O operations. These system calls provide an event notification interface that can be used to register file descriptors and receive the readiness notifications through polling.

Furthermore, interrupts can be used in lieu of polling to deliver readiness notifications to the application. For instance, the Linux AIO library supports only asynchronous access to disks through interrupts. Capriccio [214] uses this model to provide a blocking I/O abstraction on top of asynchronous calls.

I/O Polling

Blocking user-level entities along with I/O polling is used to provide synchronous blocking interface in user-level threading runtimes. Polling I/O devices requires support from the operating system. Linux and FreeBSD both support `select` and `poll` polling mechanisms. `epoll` is only supported by Linux and `kqueue` is only supported by FreeBSD.

`select` and `poll` accept multiple sets of file descriptors marked for read or write or exceptions. Developers have to initialize and fill up these sets with the file descriptors that require monitoring. When the system call is issued, these sets are copied to kernel-space. If data is ready to be read or a buffer is available for write, the kernel updates the internal status of the corresponding file descriptor. The application then has to read the sets and iterate through them to find the ones that are ready and use them for further processing. There are various problems with both `select` and `poll`that are listed in [225, 4].

The interfaces of `select` and `poll` are stateless and they both handle file descriptors in a linear way that depends on the number of file descriptors. Banga et al. [225] suggested a

new stateful event-delivery mechanism that allows the application to register interest in one or more sources of events, and to efficiently dequeue new events. The suggested approach is independent of the number of file descriptors and therefore scales better. Instead of providing the entire set of file descriptors during each system call, the kernel internally maintains the set and provides an API to add, remove, or modify file descriptors. The application declares interest in a file descriptor using a system call and the kernel updates the file descriptor set internally. The application can get new events from the kernel by calling another system call, which returns a list of file descriptors that triggered an event.

Therefore, each event is registered individually with the kernel, and the kernel only returns the set of ready file descriptors when the applications polls for notifications. Hence, the application does not have to iterate through all file descriptor and find the ready ones every time. Also, registering and deregistering file descriptors does not translate to iterating through all file descriptor and register/deregister their callbacks inside the kernel. This work inspired Linux and FreeBSD to come up with their own implementations, `epoll` and `kqueue` respectively. Both of these approaches are exclusive to their operating systems and therefore not portable.

`epoll` is the latest polling method in Linux [124] that provides a stateful and scalable solution for I/O polling. It also allows attaching additional context to each monitored event, e.g., a pointer to the object that should handle the notification, and therefore saves one additional lookup. Moreover, `epoll` allows adding and removing file descriptors, or modifying their events, even when another thread is blocked in *epoll_wait*. These capabilities provide additional flexibility and improves the interface over `poll` and `select` that do not support this feature. Furthermore, since the kernel has a list of all monitored file descriptors, it allows the kernel to register events on them, even when the application is not polling for notifications.

Nevertheless, `epoll` does not have a great performance advantage over `poll` or `select` when the number of connections are small, or connections do not have a long lifetime, since `epoll` requires one additional system call to add descriptors to the epoll set in comparison with `poll`. Choosing between `select`, `poll`, and `epoll` depends on the type of the application, number of connections, and duration of each connection. `epoll` plays very well with multi-threaded applications that handle large number of relatively long connections.

L. Gammo et al. [75] compare the performance of `epoll`, `poll`, and `select` using μserver [35]. The HTTP server is subjected to a variety of workloads and it is shown that `select` and `poll` mechanisms perform comparably well in the absence of idle connections. But introducing idle connections results in dramatic performance degradation when using `select` and `poll`, while not noticeably affecting the performance when using `epoll`.

`epoll` supports two triggering modes: *level-triggered* and *Edge Triggered (ET)*. Level-triggered mode is the default mode and is similar to how `select` and `poll` deliver notifications. In this mode, as soon as the file descriptor state changes from 'not ready' to 'ready', `epoll` delivers a notification. As long as the state stays in 'ready' mode, `epoll` keeps delivering a notification for the file descriptor every time the network is polled. However, using `poll` or `select` this is a simple bitmask operation done entirely in user-space.

In *level-triggered* mode, changing the event flags, e.g., from write to read, requires issuing a system call on all the file descriptors that require the change. `epoll` does not support batching of this operation so for a large number of file descriptors this translates to a large number of system calls. In contrast, in *edge-triggered* mode the readiness notification is delivered only once the File Descriptor (FD) is ready. *edge-triggered* mode is discussed in more detail in Section 3.4.

Since `epoll` internally uses a kernel object (the "file description") instead of directly using the file descriptor, there can be issues when *close()* is called on a file descriptor without removing the underlying file description from the kernel. For instance, calling *dup()* on a file descriptor that is registered with `epoll` and closing the file descriptor afterwards without removing it from `epoll` results in `epoll` delivering notifications for it forever.

Moreover, it is not possible to remove the file descriptor from `epoll` anymore since `epoll` only removes valid file descriptors. But since the kernel object still exists, `epoll` assumes the file descriptor still exits. However, this issue is must be avoided through careful application design. The programmer has to make sure that *close()* is always called after the file descriptor is removed form `epoll`. Finally, it makes it hard for creating abstractions on top of `epoll`, since it is not possible to forbid users from calling the *close()* themselves.

FreeBSD's `kqueue` [118] follows the same concept of having the kernel manage file descriptors internally for better scalability. Unlike `epoll`, `kqueue` makes it possible to add, remove and update multiple events with a single system call. This approach avoids the extra overhead that `epoll` imposes on the application by requiring all events to be registered individually.

Moreover, in `kqueue` the file descriptor is removed from the queue whenever the *close()* system call is called on it. Therefore, programming applications with `kqueue` is easier and the duality problem similar to `epoll`, as explained above, does not apply to `kqueue` events.

Furthermore, neither of `select`, `poll`, or `epoll` is designed to work with regular files (disk I/O). Since all these system calls have an assumption of readiness that assumes disk files are always ready and do not need to be monitored. In contrast, `kqueue` provides support for disk I/O since disk I/O can block when the data is not cached in memory.

With `kqueue` an application issues I/O operations on disk files and gets notified when they are done.

2.3.2 Scheduling

Runtime systems apply broad scheduling strategies to satisfy several objectives, such as resource utilization, load balancing, fairness, and responsiveness. Each objective enforces different requirements on the scheduler.

For instance, in a system where responsiveness and fairness is important, such as real-time systems, preemptive schedulers are employed. Preemptive schedulers assign and take away the CPU from tasks by giving each task a *quantum*. The scheduler then preempts the running task after it uses its assigned quantum. Preemption prevents starvation of tasks and provides fairness according to the requirements of the application. Moreover, the application stays responsive as all tasks get a chance to run on the CPU.

However, preemption requires context switches between tasks that can lead to bad cache locality based on what happens after the context switch. First, if a task running on a CPU is interrupted and later it is assigned to another CPU, it is possible that the data has to be fetched from the memory to the local cache of the second CPU. Second, even if the tasks get to execute on the same processor, other tasks that execute on that processor can cause the data to be evicted from the local caches. Therefore, when the interrupted task start executing again, it has to pay a penalty and reload its data from the main memory. In addition, if the time slice is too short most of the processing power is consumed by the scheduler, and if it is too long it can affect the responsiveness of the system.

In contrast, non-preemptive schedulers do not interrupt tasks and once the CPU is assigned to a task, the task has to voluntarily relinquish the CPU. In other words, tasks have to *cooperate* with each other to provide fairness. Therefore, systems that do no have strict requirements for fairness and responsiveness can benefit from *cooperative* scheduling and avoid the overhead of preemptive schedulers. Tasks have more control over when they context switch and therefore can be scheduled to context switch only when it does not hurt their cache locality. However, non-cooperative tasks can lead to starvation of other tasks or cause unfair access to the resources if not programmed correctly.

In some applications, such as High Performance Computing (HPC), and task parallelism frameworks, tasks run to completion and do not block on external events after they start executing. For instance, in High Performance Computing (HPC) applications where tasks form a Directed Acyclic Graph (DAG), child tasks run to completion after they

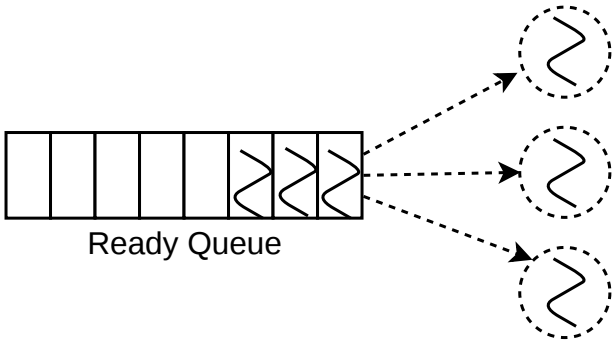


Figure 2.1: Shared Ready Queue

start executing, and parent tasks only block to wait for their children to complete. For such applications, fairness for individual tasks is not important and the objective is proper load-balancing and resource utilization along with lowering the overall execution time of the application. Therefore, cooperation from individual tasks is not necessary and parallelizing and load-balancing strategies play a more important role in improving the overall performance.

However, in applications such as network servers where fairness is of great importance to meet latency requirements, cooperation among tasks is required to avoid starvation of other tasks. For instance, if a task serving a connection does not relinquish the CPU, other tasks waiting for that CPU starve, which leads to high latency. Therefore, depending on the requirements of the application, programmers have to make sure their application provides fairness among tasks. Tasks can either voluntarily yield the CPU to other tasks or be blocked on an external event, such as I/O events.

Other than fairness and responsiveness, in multiprocessor environments schedulers are responsible for distributing and load-balancing tasks in order to properly utilize resources and improve the overall performance. Distributing tasks among processors often requires the use of single or multiple producer-consumer queues that are shared among processors. Since these queues are shared among the processors, pushing to and pulling from a queue must be performed atomically. Hence, mutex locks are often used to protect the queues and enforce coordination among the processors. In addition, lock-free alternatives can be used instead of using a mutex lock.

Usually when a producer-consumer queue is used, producers push to the back of the queue and consumers consume from the front of the queue. Therefore, it is possible to decouple protecting each end of the queue and thus push and pull operations. Hence, a queue can be protected using a single mutex lock or two separate locks that protect each

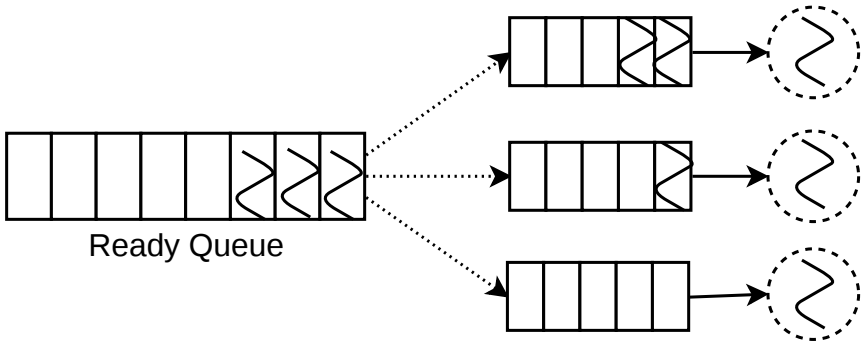


Figure 2.2: Shared Ready Queue and Local Queues.

end of the queue. Lock-free alternatives enforce atomicity of pull and push operations by decoupling them from each other. Decoupling push and pull operations can lower the overall contention over the locks.

The simplest placement strategy uses a central First In First Out (FIFO) queue, where all processors push ready tasks to the back of the queue. The queue is polled by all processors to find work and is protected by a single mutex lock. This design is presented in Figure 2.1. This design trivially provides excellent resource utilization and load balancing through work-sharing. However, the resulting contention often turns the shared queue into a bottleneck. Furthermore, this design does not support certain execution strategies, such as task to processor affinity.

In order to remove the bottleneck and make the design more scalable, it is possible for each processor to have a private ready queue. Instead of dequeuing work one by one from the central FIFO queue, each processor extracts a chunk of the available tasks from the central queue, which becomes the local queue. This design is illustrated in Figure 2.2. Processors push tasks to the central queue for load-balancing one by one. In workloads with large number of tasks dequeuing accesses are less frequent since more tasks are transferred to the local queue with each dequeue, which leads to lower contention on the central mutex. However, for workloads with a few number of tasks, this can fall back on dequeuing tasks one by one and does not have any advantage over the previous design.

This design provides good resource utilization and load balancing. However, placing tasks back to the central queue does not support affinity. Moreover, it does not scale to a large number of cores due to the central mutex still being contended. It is possible for a processor to place the tasks back on its local queue instead and use the central queue only for staging new tasks. This strategy reduces the contention over the central queue and creates strong task to processor affinity. However, this design does not let any other

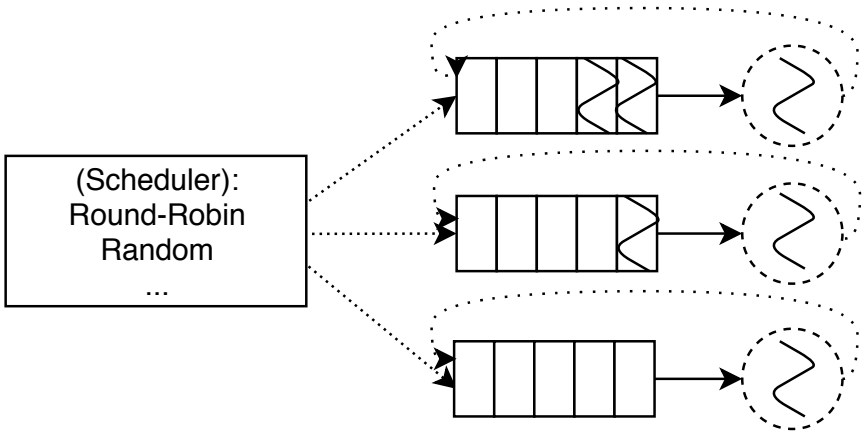


Figure 2.3: Local Queues

processor access the local queue of a processor and resume blocked tasks on that queue. Hence, providing affinity when tasks are being unblocked by other processors becomes impossible.

To address the problem above, each local queue protected by a mutex so other processors can access the local queue of each processor. The central queue provides good load-balancing through staging new tasks, and shared local queues provide a way of supporting task to processor affinity.

An alternative is to remove the central queue altogether and perform load-balancing through work-pushing by round-robin, random, or other task assignment policies. This design removes the central bottleneck altogether and can provide better scalability. Hence, each newly created task is actively distributed among other processors during the initial placement. But unblocked tasks are placed on the local queue of the processor they have affinity with. Figure 2.3 illustrates this design.

Although this design removes the single bottleneck and provides better scalability, when tasks are fine-grained and the application is running with large number of cores, the lock that protects a local queue can still become contended and slow down the application. This problem can be addressed by decoupling push and pop operations in the queue either by using two separate locks or lock-free alternatives. Since each local queue has only a single consumer, a lock-free Multiple Producer Single Consumer (MPSC) queue can be used that provides better scalability in comparison with protecting the queue with a single lock. Two variations of intrusive blocking lock-free MPSC queues are discussed in Chapter 3: Nemesis queue [40], and a stub queue inspired by [217].

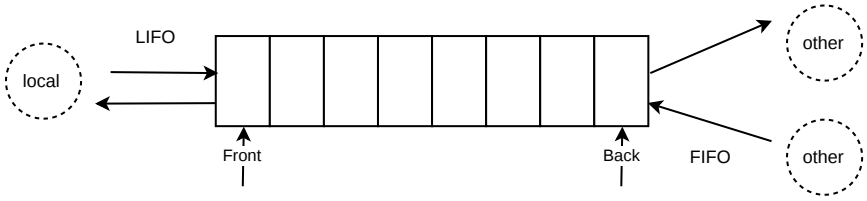


Figure 2.4: Deque Used in Work-stealing

Using a shared local queue for each processor provides better scalability in comparison with only using a central queue. In the absence of a central queue, this design works well with balanced workloads where each processor receives approximately the same amount of work and task execution times are approximately the same. However, when the workload is not balanced, some processors can be left without any tasks to run while other processors have multiple tasks queued up in their private queues. *Work-stealing* can solve this issue by letting processors steal from another processors' queue.

Work-stealing has emerged as a popular strategy for task placement and scheduling [224] in task-parallelism frameworks. Work-stealing primarily addresses resource utilization by stipulating that a processor running out of work "steals" tasks from another processor's run queue. As well, scheduling strategies based on work-stealing have also been shown to address load balancing. Classical formulation of work-stealing is limited to the fork-join pattern, but it has been generalized and investigated for general multi-threaded programs with arbitrary dependencies [28].

In work-stealing, in contrast to work-pushing and work-sharing, tasks are only migrated when a processor becomes idle, which avoids task migration until absolutely necessary. In the system described by Blumofe and Leiserson [28], each processor has a double ended queue, called a deque, where data are accessed from both ends of the queue. Every processor treats its own deque as a Last In First Out (LIFO) and take tasks from one end until it runs out of work. When that happens, the processor becomes a thief, choosing another processor as a victim, and stealing a task from the other end of the victim's deque, i.e., treating that deque in FIFO manner. However, if a scheduler apply LIFO as their primary strategy it can lead to starvation of the tasks at the end of the deque if the running tasks keep creating new tasks.

Work-stealing provides both *depth-first* execution by popping the newest tasks from the local deque, which promotes temporal and often spatial locality, and *breadth-first* execution by stealing the oldest tasks from a victim deque, which results in good load-balancing while minimizing the scheduler overhead.

The main overhead of work-stealing occurs during the stealing phase when an idle processor polls other deques to find work, which might cause interprocessor communication and lock contention that can negatively affect performance. The particulars of victim selection vary among work-stealing schedulers. In Randomized Work Stealing (RWS), when a worker runs out of work it chooses the victims randomly.

Work-stealing is used in task-parallelism applications to improve processor utilization when there is a local ready queue per processor. In addition, work-stealing improves cache locality among tasks that form a DAG and reduces the communication overhead by preventing the processor from unnecessarily distributing tasks among other processors.

Ultimately, to address the load-balancing issues with multiple local queues, the stealing phase of work-stealing can be combined with work-pushing or work-sharing to provide better load balancing. Tasks are distributed using a placement strategy as described above, and in addition, a processor can steal from the local ready queue of another processor when it runs out of work.

2.3.3 User-level Threading Libraries

In this section, existing user-level threading runtime systems are categorized based on their use-case and characteristics. In addition, different components of each system is studied and explained in details. First, user-level threading runtime systems designed for HPC applications are presented. These frameworks are designed for CPU-bound applications and either do not provide a synchronous I/O interface or provide very limited support. Since this thesis focuses on the performance of network applications, the I/O issues are studied in more detail. Second, user-level threading that support the actor model of programming are explained. Next, user-level threading libraries that are written for network applications and provide some form of synchronous I/O interface are listed. This category is further split into two parts: frameworks that only support N:1 mapping, and frameworks with M:N mappings. It is important to note that frameworks that are running over a virtual machine, e.g., user-level threading frameworks based on Java, are not considered in this work. In addition, since Java itself relies on pthreads and this document compares against pthreads, including Java does provide additional insight and information. However, managed runtimes that are compiled directly to machine code, such as Go, are studied and evaluated. Other specific purpose user-level threading libraries are listed in Appendix A.

HPC Runtimes

Cilk [27], Cilk++ [117], and Cilk Plus [100] are programming languages designed specifically for parallel computing. These frameworks provide a user-level threading library and also provided extensive support for parallel algorithms and structures. A Cilk thread is a non-blocking function that runs to completion without waiting or suspending after it starts executing. Cilk and all its descendants provide a work-stealing scheduler to provide better locality for task parallelism. These frameworks also provide synchronization primitives, race detectors, and performance analyzers. They differ by the level of support for various parallelism features. None of these frameworks have support for I/O operations as their main purpose is to scale scientific computation problems.

Qthreads [221] is a portable user-level threading library developed by Sandia National Laboratories for massive parallelism. Qthreads aim to support millions of lightweight threads.The library is designed with NUMA architectures in mind by providing a scheduling domain for grouping kernel-level threads under the same domain. Qthreads provides a simple round-robin scheduler but it has been extended to support a NUMA-aware work-stealing scheduler [147]. It also provides synchronization primitives such as a mutex. However, this library provides very little support interacting with I/O through a synchronous interface. It generally relies on asynchronously running I/O operations using a pool of kernel-level threads that serve blocking I/O operations and system calls. Hence, performance is low due to frequent transfer of control and blocking at the kernel level.

MPC [154] uses M:N user-level thread library to deal with communications and synchronizations in NUMA architectures. This framework is aimed for HPC applications and does not provide a synchronous I/O interface.

MassiveThreads [143] is focusing on scheduling recursive task parallelism along with user-level context switches triggered by I/O calls. It uses work-first work-stealing to support recursive parallelism. MassiveThreads do not use *ucontext* on Unix systems to avoid the large overhead due to internal system calls and uses context switching routines that switch callee-saved registers only. For blocking I/O, user-level threads are blocked and queued in a "blocked list" representing the file descriptor, and I/O polling is performed using epoll. The context switch performance and scheduling performance is evaluated using standard HPC workloads, but the I/O performance evaluated using a ping-pong benchmark and only against pthreads, which does not provide enough information about the performance I/O mechanism.

Argobots [171] presents a lightweight, low-level threading and tasking framework to support massive on-node parallelism. It supports two kind of work units: user-level threads

and tasklets. The former have an associated stack and allow blocking calls, while the latter does not. In addition, Argobots does not provide any smart policies, but only threading and tasking mechanisms, so users can develop their own solutions. The authors state that smart policies (e.g., work-stealing scheduling) sometimes conflict with the characteristics and demands of the application. Besides, to enable flexibility in usage as well as minimize scheduling costs, Argobots supports a notion of *stackable schedulers* with pluggable queuing strategy. It also provides mapping mechanisms between work units and each hardware resource (e.g., a core).

The major difference between Argobots and other threading libraries is that Argobots is designed to be an underlying threading and tasking runtime for high-level runtimes or libraries. Argobots is mainly designed for HPC applications and the details of the I/O subsystem are not provided. Event notification management and interaction with external blocking resources (such as the network and storage devices) are delegated to the programmer by requiring explicit context switches.

Actor Runtimes

Erlang [212] is another programming language that supports concurrency and distributed programming. Erlang follows the Actor model to implement concurrency by spawning *processes* that do not share data, and instead use message passing to communicate (These are different from system processes). Erlang relies on a shared nothing model and thus does not require synchronization and mutual exclusion.

Erlang uses epoll Level Triggered (LT) for polling network I/O on Linux, it also supports kqueue on FreeBSD, and poll on Solaris. Erlang provides an M:N mapping of user-level threads to kernel threads, where user level threads start with a small stack and the stack is increased as required. In addition to a private stack, Erlang processes (i.e., user-level threads) each have a private heap. Erlang implements a work-stealing scheduler and allows the configuration of scheduling via parameters when the program is being executed. All aspects of the scheduler, such as, busy wait threshold, sleep time, binding to underlying cores, frequency of checking for I/O events and many more options can be passed to the scheduler through command line arguments. Therefore, the Erlang scheduler can be modified to perform according to the provided workload. However, the programmer should have a deep understanding of the scheduler to be able to tailor it.

C++ Actor Framework (CAF) [46] is an open source implementation of the actor model in C++. Actors in C++ Actor Framework (CAF) are light-weight and cooperatively managed by a work-stealing scheduler. Actors communicate among each other through message

passing across cores or the network. CAF also provides lock-free data structures to be used along with the framework. CAF does not implement user-level threading and only multiplexes actor objects over kernel-level threads.

N:1 User-level Threading Runtimes

Capriccio[214] is used on high-concurrency servers to hide I/O latency, but only in single-threaded applications. It was an effort to port SEDA to C where user thread libraries are used to avoid extensive context switches by relying on cooperative scheduling. Capriccio determines an approximate bound for user-level stack size prior to compile time using static analysis. However, this approach cannot detect recursion or might be too conservative, but in addition Capriccio supports dynamic expanding or shrinking of the stack. Capriccio predicts when a stack overflow is about to happen and uses dynamic stack allocation to expand the stack size. Capriccio uses epoll for file descriptors that can be polled (sockets, pipes, and FIFOs) and Linux AIO for disk. It also provides basic synchronization primitives since it only supports a single kernel-level thread.

GNU Pth [68] provides user-level threads on a single kernel thread with emphasis on portability. It provides non-preemptive priority-based scheduling for multiple threads of execution inside event-driven applications. Pth also provides standard Portable Operating System Interface (POSIX) API replacement (such as, read, write, and sleep) and synchronization primitives (such as, mutex, read/write mutex, and condition variable).

Windows Fibres [138] provides User-Mode Scheduling [139] with N:1 mapping that provides cooperative scheduling at the user-level. Each User-Mode Scheduler has its own ready-thread queue of worker threads. Ayda et al. [6] propose a hybrid model where automatic stack management is performed by *Windows Fibres*, and event-driven programming is used as the underlying cooperative task management. Therefore, transition between functions goes through a fibre instead of multiple call backs.

State Threads [173] provide an API for writing server applications such as web servers with an event-driven state-machine architecture using user-level thread per connection. Scheduling is non-preemptive and no explicit yield is needed since sooner or later a thread performs a blocking I/O operation and gives up the control. State Threads provides N:1 mapping of user-level threads to kernel threads but provides inter-process synchronization to support many N:1 mapping rather than M:N mapping for better scalability. Coordination among processes is left to the application programmer.

Facebook Folly [179] is a C++ framework that uses fibres for parallelism. Folly provides low-level synchronization primitives (Batons) as well as high-level primitives on top of

them, such as mutexes. Each fibre has a fixed stack size and Folly provides a built-in mechanism for stack-overflow detection. In addition, functions that do not suspend the fibre can be executed on the kernel thread's stack to avoid running out of stack space. Moreover, each fibre can have an optional fibre-local data and Folly provides a GNU Project Debugger (GDB) extension for debugging of fibres. Mapping of fibres to kernel threads is N:1. Folly also provides futures that can be integrated with fibres.

libmill [191] introduces CSP concurrency to C and the interface is similar to Golang. It maps many coroutines to a single kernel-level thread, thus N:1 mapping, and coroutines can communicate through channels similar to Golang. It provides coroutine-local storage and supports non-blocking POSIX functions for network programming.

libdill [190] is a C library that makes writing structured concurrency easy. Structured concurrency means that lifetimes of concurrent functions are cleanly nested. If coroutine *foo* launches coroutine *bar*, then *bar* must finish before *foo*. libdill provides M:N mapping of coroutines but coroutines only execute on a single thread and cannot migrate, therefore N copies of N:1 mapping running in one process.

StackThreads [199] provides a low-level C library to support user-level threads only on a single processor. It forks new threads through a sequential procedure call. When the procedure blocks, it can resume its caller by moving its frame from the stack to the heap and unwinding the stack. Since the caller can be rescheduled even if the callee is blocked, it is effectively a new thread of control. The blocked thread's context can be restored on top of the stack and the control is transferred to the point where it was blocked. However, this approach prohibits taking the address of stack-allocated objects since they are accessed relative to specific registers. Also, caching the address of stack allocated variables in registers becomes problematic. StackThreads saves and restores the entire stack frame and can end up with high context-switch costs.

libco [202] provides N:1 mapping and uses an approach similar to StackThreads [199] and copies out the previously running user-level thread stack and copies in the stack of the next user-level thread for each context switch. Therefore, it is prone to the same problems as StackThreads.

M:N User-level Threading Runtimes

StackThreads/MP [198] extends StackThreads by providing thread migration on shared-memory multiprocessors. It also employs a more involved stack frame management in which frames of suspended threads are not copied but retained in the stack. Stack frames for lightweight threads are allocated on the physical stack of the worker threads and linked

to each other and other stack frames located on another physical stack. For migrating a thread, all threads above the migrating thread should be suspended first along with the migrating thread, until the control reaches the parent of the migrating thread. Other kernel-level threads constantly poll each other to find migrating threads and restart them. Although, this approach removes the limitations of StackThreads, it has relatively high overhead for managing stack frames and migrating user-level threads.

μC++ [38] is a dialect of C++ that provides advanced control-flow including light-weight concurrency on shared memory multi-processors. μC++ provides new kind of classes to support concurrency: coroutines, which have independent execution states; tasks, which have their own threads; and monitors which allow for safe communication among tasks. μC++ provides a translator that reads a program containing μC++ extensions and translates them to proper C++ statements and uses a C++ compiler to compile and link them with μC++ concurrent runtime library.

μC++ provides a M:N mapping of user-level threads (tasks) to kernel threads and clustering of user-level threads and kernel threads is also possible. The scheduling of user-level threads are performed using a round-robing preemptive scheduler. I/O management is currently done through object oriented, non-blocking I/O using `poll` and `select`. Objects in μC++ communicate by sharing memory through routine calls and mutual exclusion is implicit and limited in scope in the programming language constructs. μC++ provides static stack allocation where the minimum size of the stack is machine dependent, and is as small as 256 bytes. Stack overflow is detected by a function call and if detected, the program terminates.

Mordor [142] is an I/O library based on fibres. Scheduling of I/O fibres is done through a central fibre queue through work-sharing. Overall, the implementation is not scalable and due to lack of documentation it is hard to evaluate the performance of this library.

The authors in [78] provide high-level discussions regarding benefits of scalable M:N threading, which lacks implementation details. Also, the evaluation section does not provide the experiments specifications, and results are not justified. For these reasons, this work cannot be evaluated or compared against and is not considered a full user-level threading implementation.

C++ Boost library [57] supports multi-threaded and concurrent programming. It provides coroutines, context (cooperative multitasking on a single thread), asynchronous I/O, atomic variables, lock-free data structures, interprocess communication, and a message passing API. Boost also provides an implementation of user-level threads that are scheduled cooperatively called fibres. It supports a round robin scheduler by default, but a scheduling algorithm can be extended through the provided API. This library also pro-

vides synchronization primitives such as mutex, channel, barrier, future, and condition variable. However, it does not provide a synchronous I/O interface and the details of I/O operations are not hidden from the programmer.

Go Programming Language (golang) [79] supports concurrency by providing user-level threads that are called *goroutines*. Goroutines communicate with one another and synchronize their execution through *channels*. Channels are adopted from Hoare's CSP [97]. Go aims to provide a simple programming interface by removing complexity, and enabling safe concurrent programming. Go provides dynamic stack allocation by monitoring each function call and if a stack overflow is about to happen, the runtime allocates a bigger stack and copies the old stack to the new one and updates all the pointers that point to the old stack content.

The Go scheduler uses this stack growing mechanism to provide *partially preemptive*, i.e., it cannot preempt goroutines at arbitrary point and only at function calls. When the runtime checks whether there is enough room left on the stack for the function call, it probes whether the goroutine has been executing longer than a predefined time limit, if so it sets a flag, and when the flag is set the goroutine call a function to allocate more stack. However, instead of allocating stack the goroutine is preempted, switched out and placed in the ready queue to resume execution later.

The Go scheduler is a combination of work-stealing and work-sharing. Each kernel-thread has a bounded local, deque which is treated as LIFO by the local thread and FIFO by other threads. However, all threads share a global ready queue that is used for new goroutines and yielding goroutines. In addition, if the local queue of a kernel thread is full, the scheduled goroutine is placed in the global ready queue. When a thread runs out of work, it checks its local queue and then the global queue respectively. If both queues are empty the thread also polls the network in order to unblock goroutines that are blocked on network I/O. If no goroutine is available, the thread tries to steal from the local deque of another thread. Go provides a synchronous I/O interface based on epoll ET, which uses the same basic mechanisms, but a different strategy, as the approach presented in this thesis.

Table 2.1 lists the M:N user-level threading frameworks that provide support for I/O. The table lists various characteristics of each system, such as scheduling, I/O polling, memory model, and etc.

Table 2.1: M:N user-level threading frameworks with I/O support

Library	Mapping	Scheduling	I/O	Memory model	Memory	Stack	Context
μC++	M:N	Preemptive	poll/select	Shared Memory	Unmanaged	Static	General purpose
Mordor	M:N	Cooperative	epoll	Shared Memory	Unmanaged	Static	Network Servers
Boost Fiber	M:N	Cooperative	N/A	Shared Memory	Unmanaged	Static	Concurrency
Golang	M:N	Partially Preemptive	epoll ET	Message passing (CSP) & Shared Memory	Managed	Dynamic	General purpose
Erlang	M:N	Partially Preemptive	epoll	Message passing (Actors)	Managed	Dynamic	Soft real-time systems

Other Programming Languages and Frameworks

OCaml [163] is a functional programming language with support for concurrency. However, OCaml currently uses a global interpreter lock to provide threading and does not support parallelism. Multicore OCaml [145] is an effort to support shared memory parallelism, which is an ongoing project.

Haskell [91] is a functional programming language that provides parallelism and concurrency through the Glasgow Haskell Compiler (GHC). The programmer can develop concurrent applications by using features, such as forking and killing threads, sleeping, synchronized mutable variables, and software transactional memory by using GHC. Haskell uses functional programming, which has different characteristics than imperative programming and is difficult to compare against. Furthermore, according to the TechEmpower web framework benchmark suite [201], none of the Haskell web frameworks, such as Servant [90] or Yesod [176], perform anywhere close to top performing web frameworks in other languages. Hence, although Haskell supports user-level threads, it is not studied in this document.

Rust [3] does not support user-level threads to avoid the complexities of supporting a larger language runtime to manage threads. Hence, Rust only supports 1:1 threading and provides other means of managing concurrency at the user-level.

Chapter 3

User-level Threading Runtime

General purpose Unix-based operating systems such as Solaris 2.6 and NetBSD 5.0 supported M:N mapping of user-level threads to underlying kernel-level threads (or scheduler activations) [209]. This design was abandoned due to complexity problems at the time and a change of perspective [189]. First, asynchronous signal handling support for user-level threads was deemed to be complex. Moreover, automatic concurrency management and building an M:N implementation that delivers acceptable performance across a range of applications proved to be hard and not as beneficial as originally imagined. Finally, kernel-level thread implementations, that previously had low performance and scalability, started to become performant and scalable. At the same time, the worker-pool model was adopted to simplify concurrency and address scalability issues. This model changed the way developers created applications, which involved only indirect usage of threads. At the operating-system level, support for user-level threads was complex and required application refactoring to provide better scalability. Therefore, operating systems switched to use libraries with 1:1 mapping [203].

Moving to a 1:1 threading system was considered reasonable for a range of applications with limited concurrency, and the programming languages used to write these applications had little or no concurrency support. However, this move did not facilitate applications requiring significant concurrency or required overlapping communications, e.g., computations such as High Performance Computing (HPC), web servers, and applications based on the actor model, but there were few computers at the time with significant parallelism that could support these applications.

As a result, such applications relied on some user-level frameworks explicitly developed for this purpose. For instance, HPC applications used frameworks such as Cilk[27] that

provided an efficient multithreaded runtime system with work-stealing scheduler designed with HPC use-cases in mind. Many actor-based models were also relying on their own user-level threading runtime such as Erlang [212]. However, network and web servers took a totally different path.

As time progressed, the amount of parallelism increased significantly across all aspects of the hardware stack. This parallelism forced new challenges, such as the C10K problem [107] where servers are required to handle tens of thousands of clients simultaneously to handle expanding web usage. However, at the time, connection-per-thread based approaches using 1:1 threading failed to deliver performance. The heavyweight nature of kernel threads prevented large concurrency. Therefore, developers and server architects turned to event-based programming as an alternative. Event-based solutions have a low memory footprint and can multiplex tens of thousands of concurrent connections. In addition, these early web servers were mainly running on a single processor using a single thread, so there was no requirement to deal with concurrency.

Nevertheless, the event-based programming model comes with disadvantages, which are discussed in the previous chapter. Specifically, event-base programming adds to the complexity of a program since events are handled through asynchronous call-backs. Furthermore, all event-based web-servers now use threads to handle non-blocking network traffic, i.e., there is a single-threaded event-based front-end, which eventually passes control to a threaded back-end to prevent blocking the event thread. Despite its disadvantages and hybrid approach, event-based programming is the main programming model for developing web servers due to performance considerations, e.g., Node.js [206] and Nginx [161].

Given the complexity of large event-based web-servers and their reliance on back-end threading, researchers are revisiting thread-per-connection web-servers as an alternative. However, this web-server reexamination forces a threading reexamination, because large numbers of kernel threads are still an issue. (Section 4.6 shows a large number of kernel threads is possible.) The threading reexamination is user-level threads, which provide synchronous programming for simpler applications development, transparent scalability, and support CPU and I/O bound workloads.

Hence, the thesis of this document is to revisit user-level threading and M:N mapping of user-level threads to kernel-level threads. A complete user-level-threading runtime is presented, consisting of multiple components: *scheduling*, *synchronization*, and *I/O subsystem*. This runtime is then used for performance studies on network servers that handle large concurrency. As well, other highly-threaded situations, such as actors, are tested to validate the thesis. The objective is to

- Determining and implementing the minimum required functionality to support real-

world multi-threaded applications with an initial focus on network servers.

- Simplicity and conciseness, while demonstrating the performance potential of user-level threading in comparison with other runtimes, such as event-based execution. Each component is kept as simple as possible while still satisfying the requirements of the target applications. However, various features of the runtime can be modified at compile time using provided configuration parameters.
- Providing a synchronous I/O interface and therefore, avoid the complexities of asynchronous programming.
- Vertical scalability for network server applications that need to handle millions of concurrent connections and requests. In particular, the scheduling and I/O components need to scale well across a large number of cores over multiple NUMA nodes.

The runtime can be built and executed on unmodified Linux and FreeBSD systems using the x86-64 architecture [83, 60]. It is implemented in C++ and supports both GCC and Clang compilers. The runtime provides a C API that mimics the POSIX *pthreads* API [21]. In addition, the runtime can be compiled to a static or a shared library that applications link against.

3.1 Overview

Three main concepts form the building blocks of the user-level threading runtime: *fibre*, *processor*, and *cluster*. User-level threads are called *fibres* and are the smallest execution contexts representing runnable tasks. Fibres are executed collectively by kernel threads represented as *processors*. A scheduling domain, called *cluster*, provides the familiar M:N scheduling model to execute fibres using a set of *processors*.

The runtime library provides a typical thread-based programming interface. An application can create and combine *fibre*, *processor*, and *cluster* objects to facilitate a desired execution layout. Figure 3.1 illustrates how fibres, processors, and clusters relate to each other. There are three clusters with different number of processors. Each cluster manages a set of fibres that are executed by the processors in that cluster. Fibres can also migrate to another cluster.

Processors that belong to a cluster stay in that cluster and follow the same scheduling policy and, unless a fibre explicitly asks to be migrated to another cluster, fibres stay

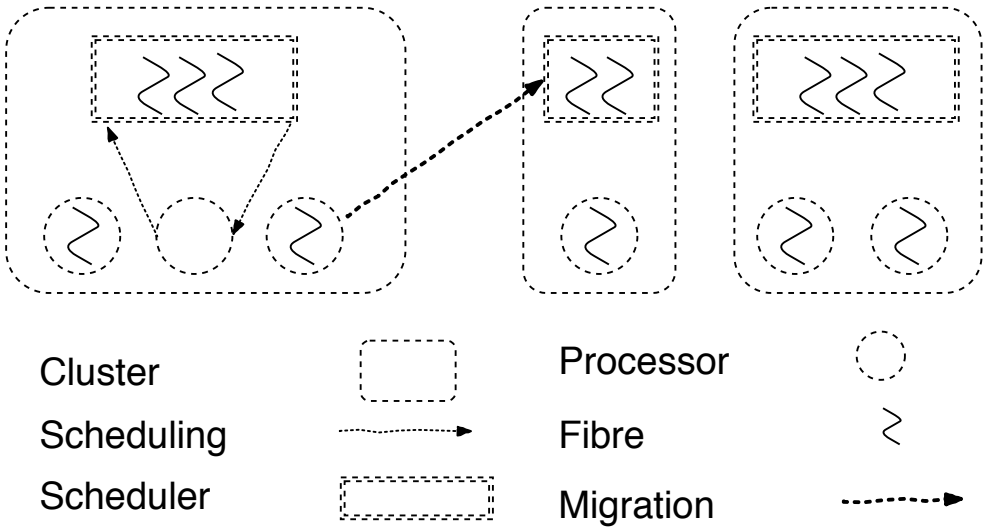


Figure 3.1: An example of a setup with 6 processors and 3 clusters

in a cluster as well. Each cluster provides its own I/O multiplexer along with the data structures required to handle the polling operations. However, the I/O subsystem is global and all File Descriptors (FDs) are globally defined and used by the polling mechanism interacting with the operating system.

A cluster provides an abstraction for a group of processors and can be used for purposes such as NUMA-aware scheduling and pipelining. For instance, multiple clusters can be created to represent different stages of a pipeline, and fibres can migrate among clusters to move along the pipeline.

Scheduling, synchronization, and I/O are the main building blocks of the runtime. Scheduling is required for efficient resource utilization in concurrent applications. Synchronization primitives are required in applications with shared mutable state. Network applications interact with I/O and therefore an I/O subsystem is required to support such applications. The performance of these runtime components can directly affect the performance of the application. Also, user-level synchronization primitives are used in the runtime, and therefore they can affect the performance of the runtime. Performance and scalability of the I/O subsystem affects the performance of network servers. Each component is briefly introduced in this section, and more detail is provided in the following sections.

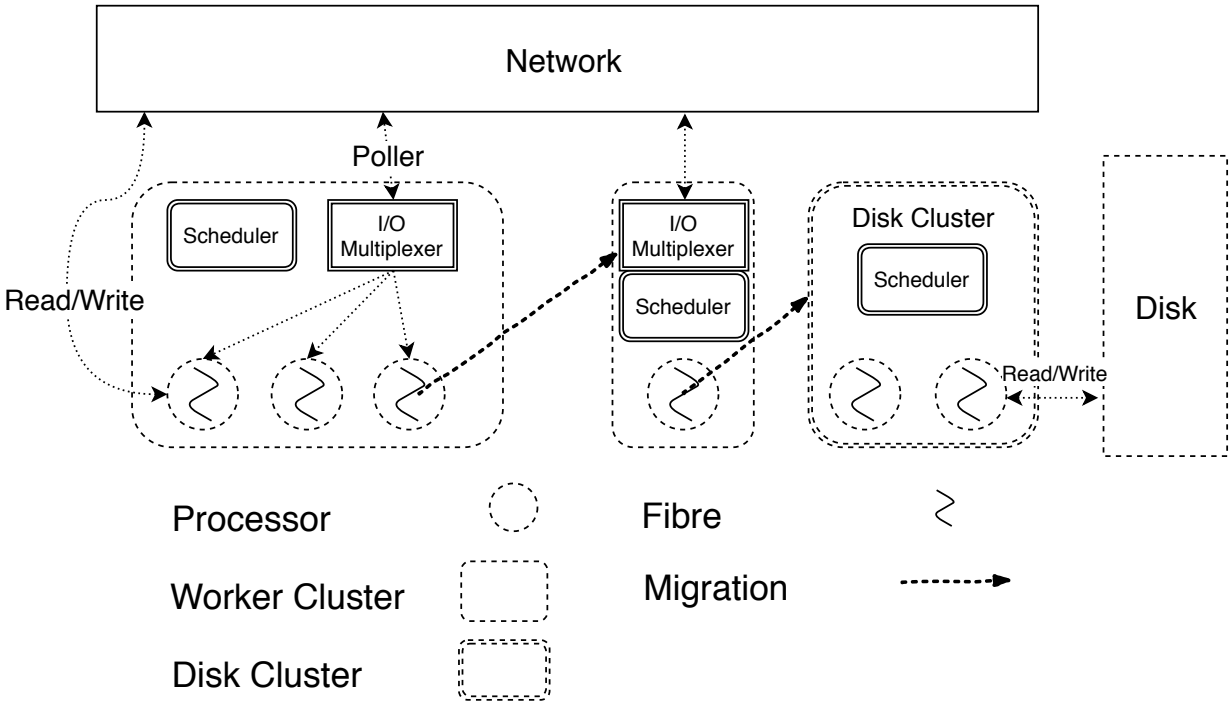


Figure 3.2: Scheduler and I/O subsystem per cluster.

Scheduling is fully cooperative and the runtime does not provide any form of preemption. The primary reason is simplicity. Supporting preemption complicates the implementation and structure of the runtime and adds overhead. In addition, a cooperative scheduler avoids reentrancy problems with unmodified libraries. Because all fibres belong to a single application, developers have other means to ensure responsiveness without relying on preemption as a resource arbiter. For instance, fibres that are serving network requests at some point yield or block on I/O and relinquish the processor to another fibre (see Section 3.4.2). In Section 3.2 the scheduling component is discussed in more detail.

User-level synchronization primitives are a necessary part of any multi-threading runtime and are supported in the runtime. Basic customary synchronization mechanisms, such as mutex, condition variable, semaphore, and read-write-lock, are provided with timeout support. These primitives are discussed later in Section 3.3.

The I/O subsystem provides a synchronous interface to interact with the underlying network connections. The multiplexer translates synchronous user-level operations into non-blocking system operations. I/O multiplexing is based on *epoll/kqueue* and interacts

with an internal event engine for the set of globally defined FDs. Each cluster has its own I/O multiplexer that relies on an instance of epoll/kqueue to asynchronously poll the network. Disk I/O operations are delegated to be performed by processors on a dedicated disk cluster asynchronously. The I/O subsystem is discussed in more detail in Section 3.4.

Figure 3.2 demonstrates the structure of an application with 2 clusters along with their components. Each cluster has a scheduler and an I/O multiplexer as explained earlier. The scheduler is responsible for distributing the fibres among the processors in the cluster and satisfying the scheduling objectives of the application. The I/O multiplexer polls the network asynchronously, unblocks the blocked fibres, and places them on the appropriate ready queue.

3.2 Scheduling

Scheduling is one of the main components of any threading runtime. The scheduler manages a set of tasks ready for execution, and assigns tasks to processors. Depending on the characteristics of the application and other scheduling objectives, schedulers are responsible for load balancing and task placement, can change the order of execution, and are preemptive or cooperative. The scheduler presented here is fully cooperative and always uses FIFO ordering to execute the tasks. However, various task placement strategies are supported by the runtime and are discussed in this section.

3.2.1 Placement

One of the main roles of the scheduler is distributing fibres among processors in a cluster. The scheduler needs to make a decision about fibre placement either when a fibre is initially created, or when a blocked fibre is being unblocked and resumed. Typically, performance of multi-threaded programs depends on good resource utilization, and therefore the first objective of the scheduler is proper utilization of resources. However, resource utilization depends on load-balancing, which introduces scalability challenges as the number of cores increases. These challenges include the extra overhead from coordinating among threads and locality problems. Schedulers can improve locality by taking dependencies among fibres and also fibre-to-processor affinity into account. Hence, fulfilling affinity and improving locality is the secondary objective of the scheduler.

Fibre-to-processor affinity is only satisfied if a blocked fibre is unblocked on the queue of the same processor where it was previously executing. Depending on the characteristics

of the workload, fibre-to-processor affinity can improve locality, and hence, increase performance of the application. Affinity is considered a form of forced placement where fibre placement is performed independent of the distribution of other fibres.

The runtime supports work-sharing, work-pushing, and work-stealing strategies, as explained in Chapter 2. These strategies can also be combined with each other using compile time flags. A blocked fibre is always unblocked on the local queue of the processor it has affinity with, if applicable.

3.2.2 Scheduler Design

The default fibre scheduler is illustrated in Figure 3.3. It manages the set of ready fibres in the scope of a cluster. All logical queues are arrays of basic queues to support fixed static priorities for workloads that have multiple levels of priority. The runtime specifies a maximum level of priority that determines the number of basic queues in each logical queue. Pushing to a logical queue pushes the fibre to the basic queue in the related priority level. Popping from a logical queue, requires iterating through the array of basic queues in order of priority and checking for ready fibres. The *staging queue* is used to stage new fibres, as well as for work sharing and migration. Fibre-to-processor affinity is satisfied by unblocking a blocked fibre on the processor it has affinity with.

In addition to the staging queue and local queues, the scheduler supports a background queue per cluster. The background queue is the last queue checked after a processor runs out of work. Fibres that do not perform time-sensitive tasks or have very low priority can be placed in the background queue. The background queue can also be used to manage the load in the system. For instance fibres that are the main source for creating other fibres, e.g., listener fibres, can be placed in the background queue to make sure new fibres are only created if the system has the capacity to serve them.

A processor primarily executes fibres from its local ready queue. If there is no work in the local queue, the processor queries the staging queue in the cluster. If no staging work is found, the scheduler attempts to steal a fibre from the processor that has been busy for the longest time. Finally, if necessary, the cluster's background queue is queried for work. The default queue data structure is a lock-free single-consumer Nemesis queue [40] with an additional lock to synchronize multiple consumers during work-stealing.

When a fibre is stolen, executed by the thief processor, and later blocked on the thief processor, it can be unblocked either on the local queue of the thief processor or the victim processor. The former provides better overall load-balancing and the latter provides better fibre-to-processor affinity. By default, the scheduler supports *transient* work-stealing, where

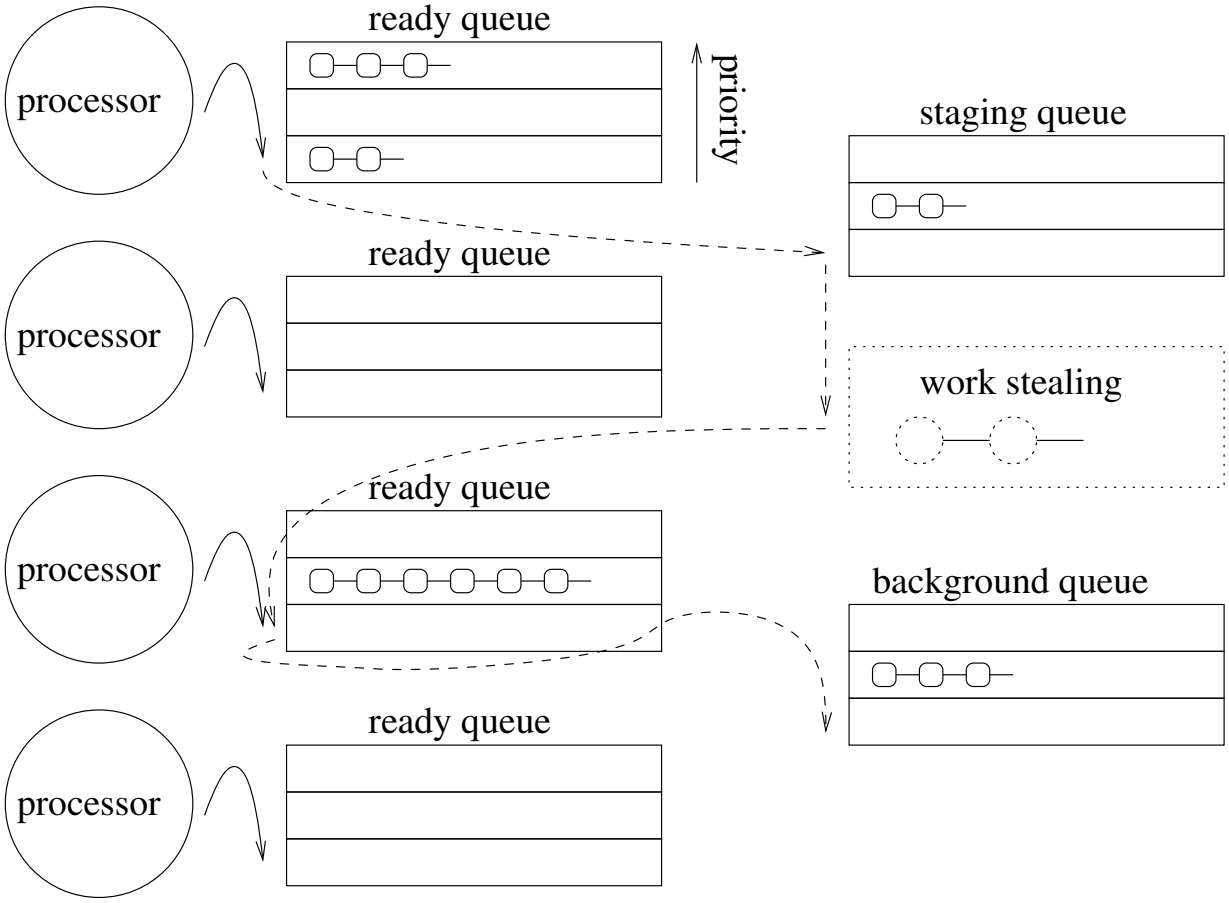


Figure 3.3: Cluster Scheduler

the scheduler moves the stolen fibre back to the victim processor. However, a *sticky* option is also provided where the stolen fibre stays in the ready queue of the thief processor. The sticky version defines a threshold to determine whether to keep the fibre on the ready queue of the thief processor or not. When a processor is stealing, it checks the size of the victim's queue, if it is less than the threshold, the fibre is returned to the original processor after execution. Otherwise, the thief processor places it on its own ready queue.

Aside from supporting affinity and priority, the basic philosophy behind this scheduler design is trading off fairness and latency for simplicity. Within the scope of a single application, it does not matter so much which task is executed, as long as some useful work is performed efficiently. However, details of this scheduler can be varied. For example,

different queue data structures along with the scalability and efficiency of the default scheduler are evaluated in Chapter 4.

3.2.3 Idle Management

In addition to distributing tasks, the scheduler keeps track of busy and idle processors in each cluster using a busy and an idle list per cluster. A processor is either in busy or idle state and is added to one of the lists accordingly. A processor is idle when its local queue is empty and it cannot find any other fibre to execute. If a processor is not idle, it is in the busy state executing a fibre or looking for a fibre to execute. The runtime also keeps track of the number of idle and busy processors per cluster.

After a processor goes idle, it places itself on the idle list and blocks on an internal semaphore. When a fibre is scheduled by the scheduler, if it is placed on the local queue of an idle processor, the processor is unblocked by signalling the semaphore. Otherwise, if there is at least one processor is idle and the fibre is placed on one of the global queues in the cluster, the processor at the front of the idle list is unblocked. After a processor is unblocked, it removes itself from the idle list.

The busy list is used during work-stealing to determine the candidate processors to steal from. Polling the empty queue of an idle processor does not result in stealing any fibres. Therefore, processors only poll the ready queue of busy processors. This is done by iterating through the list of busy processors of a cluster.

Immediately blocking the kernel-thread and unblocking it later adds overhead and might cause performance degradation. An alternative to blocking is to spin in a loop and wait for more work to become available. For instance, kernel threads in most user-level threading libraries written for HPC do not block at all and keep spinning until more work arrives or they can steal work from other threads. However, spin loops waste CPU cycles and prevents other threads from using the processors. This approach is a suitable strategy for HPC applications that usually keep a dedicated set of cores busy. However, it is not an acceptable scheduling regime for general-purpose multi-threading applications that operate in a shared computing environment, especially when also considering power conservation.

An alternative is instead of just blocking or just spinning, spin for a while before blocking. In this way, if work becomes available during the spin time, the blocking and unblocking overhead is avoided. Ultimately, if no work becomes available, the processor blocks and thus yields the CPU to other threads and avoids wasting CPU cycles. However, the spinning duration should be chosen carefully since spinning for too long or too little can cause the same problems as explained earlier.

The spinning duration depends on the type and characteristics of the workload. For instance if the inter-arrival time of new tasks is smaller than the time it takes to block and unblock the underlying kernel thread, then it is better to spin before blocking. Otherwise, it is better to block and leave the resources for other threads to use.

However, as explained earlier, since the runtime maintains multiple queues (i.e., local ready queue, a global staging queue, and a background queue), when a processor goes idle it loops through these queues before it is blocked. Moreover, with the work-stealing policy, processors additionally check the queues of busy processors before going idle. Therefore, processors are not being blocked immediately after they run out of work, and in fact they spin looking for work in other queues. These additional checks can be considered as an alternative to spinning.

Therefore, although the runtime provides the option to enable spinning before blocking, the default configuration of the scheduler does not enable extra spinning. By default the processor is immediately added to the idle list and blocks if it cannot find any fibres in any of the queues.

3.3 Synchronization

Synchronization is necessary for coordination and mutual exclusion in concurrent applications. Blocking operations are essential to support synchronization primitives such as condition variables and mutex locks. In concurrent applications with user-level entities blocking at the user-level is needed for efficiency of the application. Therefore, the runtime should support blocking at the user-level by providing efficient synchronization primitives.

The user-level runtime provides a counting semaphore, binary semaphore, mutex, read-/write mutex, barrier, and condition variable with user-level blocking. All blocking interfaces support timeouts, where the fibre is unblocked after the provided timeout. The method of coordination among fibres to access the underlying structure is configurable. For instance, a semaphore by default uses a system lock (pthread_mutex in Linux) to coordinate among fibres to access the underlying blocking queue. However, the type of this lock can be easily changed at compile time.

A mutex is often used more extensively than the other primitives and therefore has more impact on the performance of the application. A blocking user-level mutex provides multiple design choices. The difference between the design choices are widely studied for system level mutex [56, 103, 30]. It has been shown that the performance of different designs is tied to the workload/platform combination [56]. This thesis does not aim to

provide a comprehensive study of various design choices for different workload types, but rather present a few basic design alternatives to provide a general purpose mutex with acceptable performance.

When a thread tries to acquire a lock and the lock is held by another thread, the thread can spin or block waiting for the lock to become available. Spinning wastes CPU cycles and inhibits concurrency as the CPU cannot be used by other threads. In contrast, blocking synchronization requires runtime support, but releases resources for other work (within or outside the application), which is essential for overall system efficiency. If blocking and unblocking does not have any overhead, blocking the thread does not inhibit concurrency and leaves the CPU to be utilized by other threads. However, since in reality lock operations along with blocking and unblocking always have some overhead, this extra overhead can be masked by spinning before blocking using a spin-block mutex.

The mutex design alternatives provided by the runtime create a *barging* spectrum as illustrated in Figure 3.4. According to Buhr et al., "Barging occurs either by placing newly arriving threads ahead of waiting threads (cutting in line) or bypassing all waiting threads to receive immediate service."[39]. On one side of the spectrum is spin locks where a fibre waiting for the lock is always barging and contends with other fibres to acquire the lock. However, a mutex that uses *baton passing* to transfer the ownership, does not provide no room for other fibres to barge.

Depending on the scheduling policy and granularity of the tasks, when a fibre is unblocked and the ownership is transferred to it, it might stay in the ready queue for a long time before it is picked up by a processor and executed. During this time other fibres that gets a chance to execute and need to acquire the lock cannot access the lock and are suspended. These fibres have to wait in the blocking queue before they can execute again. In fact, one or all of these fibres could acquire and release the lock by the time the unblocked fibre get a chance to execute.

This problem is illustrated in Figure 3.5. Figure 3.5a shows a running fibre owns the lock, one fibre is blocked waiting for the lock to become available, and there are two other fibres in the ready queue that need to acquire the lock when executed. When the lock owner releases the lock, it passes the ownership to the blocked fibre and places it on the ready queue as shown in Figure 3.5b. However, passing the ownership directly to the blocked fibre means the waiting fibres cannot acquire the lock when executed and should be blocked as illustrated in Figure 3.5c. Although the unblocked fibre can be placed at the head of the queue to resolve this issue, the same problem still exists if there are multiple processors and the currently running fibre takes a relatively long time to execute after releasing the lock. Therefore, baton passing can introduce unnecessary delays in the application.

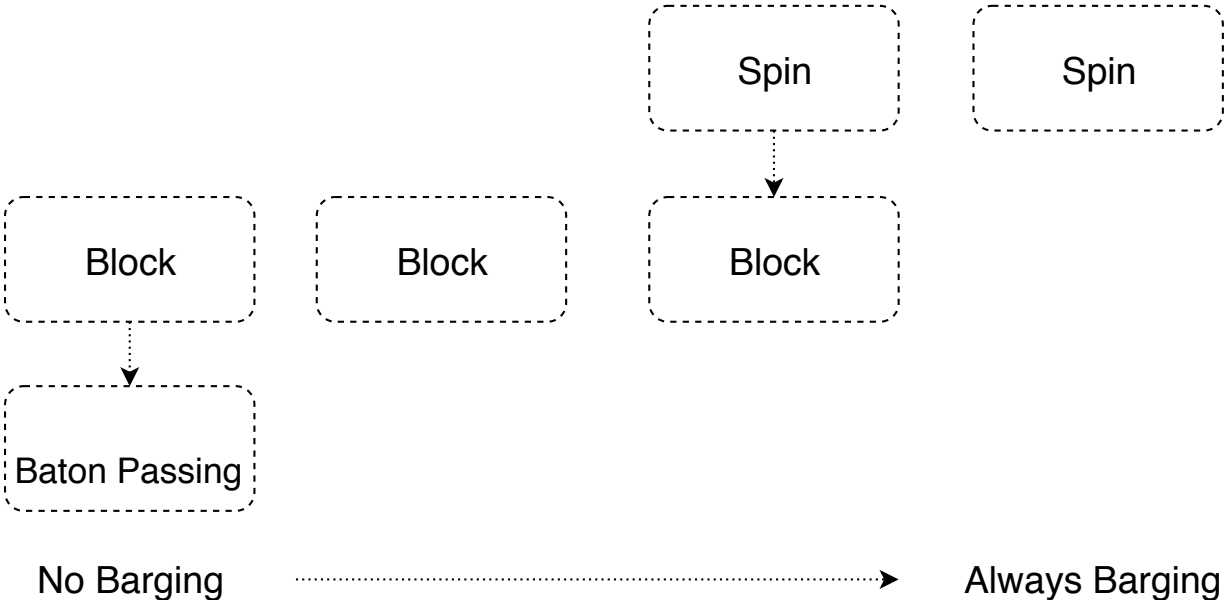


Figure 3.4: Barging spectrum for different locks.

By removing the baton passing mechanism, a blocked fibre is unblocked when the lock becomes free without transferring the ownership. When unblocked, the waiting fibre has to acquire the ownership of the lock when it is unblocked. Hence, there is more barging opportunity for the unblocked fibre and other fibres that are trying to acquire the lock in comparison to the previous design. However, this design can lead to starvation if the unblocked fibre cannot acquire the ownership of the mutex and is blocked multiple times.

It is possible to add more barging power to each fibre by spinning for a while before blocking. Therefore, when the lock is held, fibres do not give up immediately and try to acquire the ownership though spinning before they are blocked. This provides more barging opportunity whenever the waiting fibre is scheduled. However, this means a configuration parameter is required to determine the spinning duration before blocking. This topic is widely studied [56, 103, 30]. Boguslavsky et al. [30] use an analytical model and simulations to find the optimal spinning time for mutex locks. They show that the spinning duration depends on the lock holding time, context switching time, and the computing time in non-critical sections. Therefore, the optimal spinning duration varies for different workloads. Another approach is using an adaptive mechanism where the runtime adapts to the application requirements and dynamically updates the spinning duration. Such a

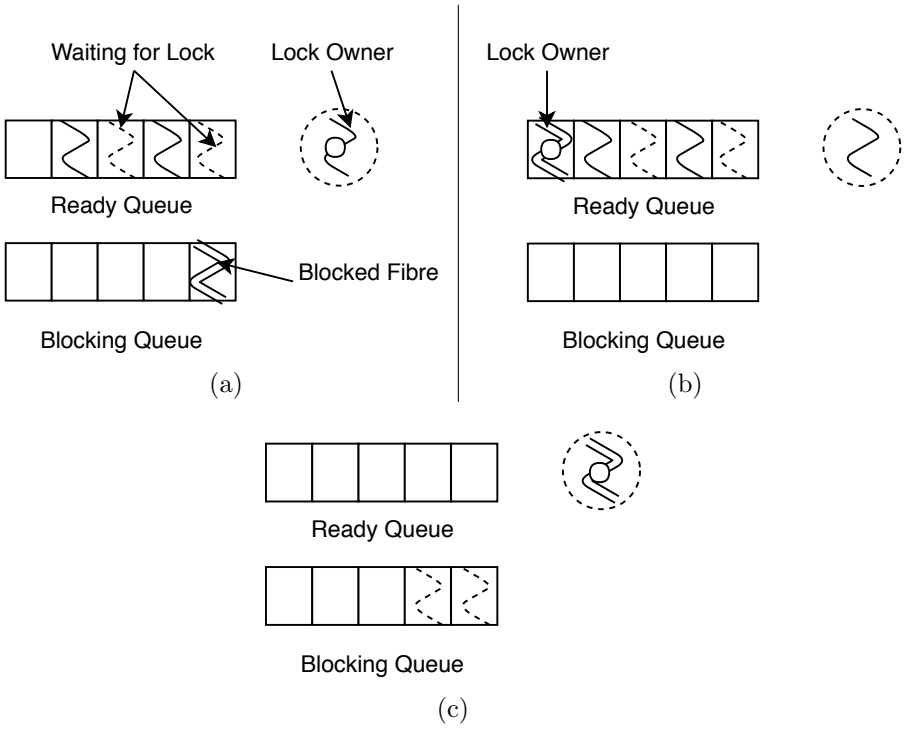


Figure 3.5: Blocking mutex with baton passing.

mechanism is implemented for pthread mutex locks.

An alternative design is to remove the blocking altogether and use a spin lock instead. Hence, the fibres are constantly trying to barge their way into acquiring the ownership of the lock. However, user-level spin locks are susceptible to the problems stated in Chapter 2. Preempting a processor that is running a fibre which holds a lock can lead to other processors that are waiting for the lock to spin for a long time. This issue arises since the operating system is not aware of the operations at user-level.

Indeed, Spinning before blocking can lead to better performance in special cases and baton passing can avoid starvation of fibres. However, since the goal of this thesis is to provide a simple and efficient design, the default user-level lock uses only blocking. The runtime provides compile time switches to change the mutex design to any of the designs or a combination of them. These design alternatives are evaluated using *Memcached* in Chapter 4.

In principle, all these design alternatives can be combined together, for instance, Go uses a hybrid solution where fibres spin for a while before blocking. If a fibre is blocked

on a mutex, by default it is unblocked without transferring the ownership. However, if the fibre that is being unblocked has been blocked longer than a provided threshold, the mutex falls back on baton passing to make sure the fibre is not starving. Hence, all running fibres can contend for the mutex unless they are preventing a fibre from making progress. For the sake of simplicity of this runtime, the investigation of more complicated locking schemes is deferred to future work.

3.4 Synchronous I/O Interface

The I/O subsystem is an important part of any application that interacts with network or disk, specially for large-scale network application servers. Therefore, the I/O subsystem satisfies the following objectives:

- Synchronous blocking interface: the I/O subsystem provides a synchronous interface where fibres are blocked at the user-level, if the runtime has to wait for I/O interfaces to become ready.
- High throughput and low overhead: one of the goals of the runtime is to show the potential of user-level threading for network application servers. To achieve high throughput and performance on par with event-based system, the overhead of I/O interactions is kept as low as possible.
- Low latency: The I/O subsystem is designed to avoid long-tail latency and provides overall low latency. This requires good application design in addition to a good I/O subsystem to provide fairness among fibres. Also, fibres should not stay blocked in the I/O subsystem for a long time when I/O becomes ready.
- Scalability: The I/O subsystem scales up as the number of cores increases.

To achieve these objectives, the I/O subsystem is designed from scratch and is configured according to throughput and latency observed from benchmarks and experiments.

3.4.1 IO Subsystem

Synchronous Interface

Providing a synchronous interface for an I/O operation requires potentially blocking a fibre that performs the operation. Blocking the underlying processor using blocking system calls

is not efficient, as it prevents other fibres that are in the ready queue of that processor from executing. Hence, non-blocking system calls should be used to perform I/O operations. If the result of the operation indicates that the underlying connection is not ready, the fibre that is executing the system call should be switched out and blocked waiting for the connection to become ready.

However, to unblock a blocked fibre, an asynchronous polling mechanism is required to poll the devices frequently and unblock the fibre when its connection becomes ready. This procedure should be handled by the runtime and be transparent to the programmer. Therefore, the runtime should provide a scalable and low overhead I/O subsystem to interact with the underlying polling mechanism provided by the OS. The runtime provides this access through three types of interfaces:

1. Most I/O operations are encapsulated by a generic wrapper that facilitates user-level blocking. The interface provides a wrapper around the familiar system calls to interact with I/O. These interfaces appear blocking to the user, but are translated to the same non-blocking operations internally. If non-blocking operations are not successful, the runtime blocks the fibre and waits for the connection to become ready by polling via *epoll/kqueue*. When the connection becomes ready, the fibre is unblocked and placed in a ready queue.

2. Specific I/O operations, for example those that need to register a new File Descriptor (FD) with the event engine, are encapsulated via specific wrapper functions.

3. If I/O operations are not supported by OS-level polling mechanisms, such as disk I/O on Linux, fibres can execute such an operation via another generic wrapper that transparently performs the corresponding system-level operation on dedicated system threads, so that its system-level blocking does not affect the execution of other fibres. The runtime creates a separate cluster to execute disk I/O operations. Before running a disk I/O operation a fibre can migrate to this cluster and transparently use one of the dedicated system threads to execute the I/O operation.

3.4.2 IO Polling

As explained in Section 2.3.1, `epoll` in Linux and `kqueue` in FreeBSD are the newest polling mechanism added to these operating systems. Both mechanisms are designed to address the shortcomings of `select` and `poll` interfaces. This section focuses on `epoll` in Linux but the findings also generally apply to `kqueue`. `epoll` is shown to have better performance

than both `poll` and `select` when there are many idle connections [75]. However, in *level-triggered* mode, `epoll` does not provide any advantages over `select` or `poll` due to additional system calls required to register File Descriptor (FD)s with the kernel.

In edge-triggered mode, however, only a single notification is delivered when the file descriptor state changes from 'not ready' to 'ready', and `epoll` does not deliver new notifications until the status changes from 'ready' to 'not ready'. Therefore, the application has to read from or write to the file descriptor until the status of the file descriptor changes to 'not ready'. Then as soon as the state changes back to 'ready', `epoll` delivers another notification. The additional system calls are avoided since `epoll` is re-armed by the kernel when the underlying socket is blocked. Therefore the overhead of system calls to switch among these operations can be avoided as well. An application that employs the edge-triggered mode should use non-blocking file descriptors to avoid blocking read or write.

Edge-triggered mode is less forgiving for programming mistakes since missing one event can cause a connection to not to be served for a long time. In order to keep track of notifications, normally the application is required to track the status of each file descriptor at the user-level to avoid stalls in the application. The infrastructure used to keep track of notifications per FD should be low overhead and coordinate among the fibres that are accessing the same FD for both input and output operations. While there are libraries providing a uniform interface across diverse OS polling mechanisms [133, 116, 2], a direct implementation ensures the greatest clarity for assessing the efficiency of this subsystem.

FDs typically have global scope across system threads in a process, thus the corresponding user-level synchronization objects are stored in a global vector, indexed by the FD value. The POSIX standard guarantees that FDs are allocated as the lowest possible integer numbers. Synchronization is provided separately for input and output. Each side is comprised of a lock, used to serialize access from multiple fibres, and a semaphore that is used to signal I/O readiness. This synchronization scheme is illustrated in Figure 3.6.[1] An asynchronously running *poller* loop interacts with the underlying OS mechanism to determine the readiness of FDs for input and/or output. In principle, the association between FD, poller, and cluster can be arbitrary. However, a straightforward layout uses one poller loop per cluster and registers each FD with the corresponding poll set. The runtime system performs all necessary interactions with epoll/kqueue and transparently handles the complexities of edge-triggered notifications.

[1] Lock icon by 'Yannick'; Semaphore icon by 'Freepik'. Both from www.flaticon.com, licensed by CC 3.0 BY.

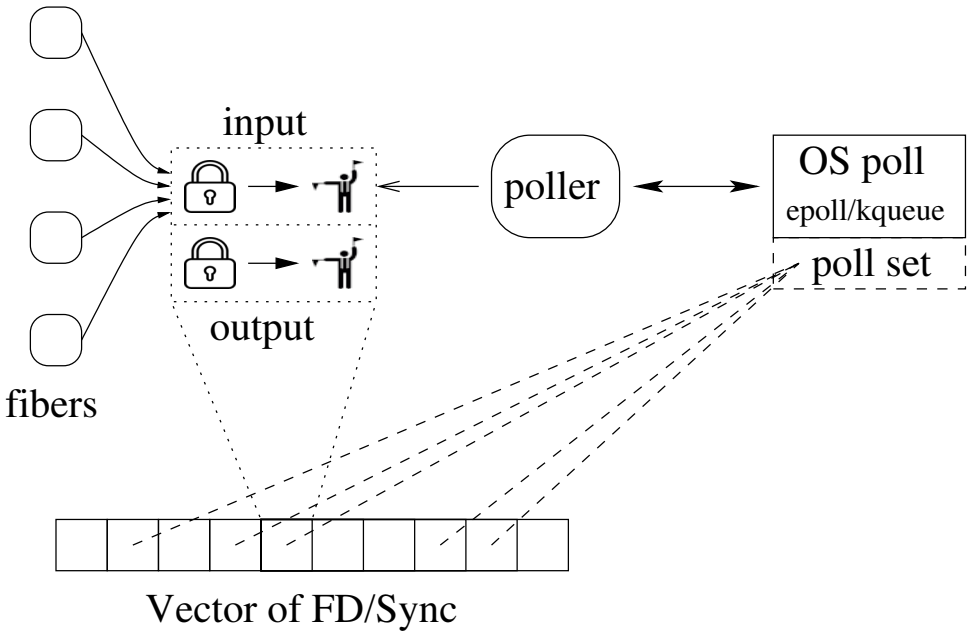


Figure 3.6: User-Level I/O Synchronization

Spurious Poll Notifications

Both `epoll` and `kqueue` share a peculiar behaviour for edge-triggered notifications – most likely a result of those mechanisms having been added to established I/O subsystems. Poll notifications are not only generated when the status of a buffer actually transitions between empty, non-empty, and full, but also when the readiness of an FD is updated in the kernel after each input or output activity. Thus, any I/O activity for an FD potentially results in a new notification for that FD, delivered during the next call to `epoll` or `kqueue`. In other words, edge triggering only suppresses notifications, if there is no I/O activity for a particular FD between subsequent poll queries. This behaviour does not significantly affect polling that is executed synchronously when an event loop runs of out work. However, it results in an increase of spurious poll notifications, if the poller loop is executed asynchronously. This behaviour is addressed by using a *binary* semaphore for user-level blocking, so that spurious notifications are largely ignored. However, this behaviour still causes overhead, but a corresponding kernel-level investigation and potential mitigation is left for future work.

Yield Before Read

Both kernel polling mechanisms and user-level synchronization facility that interacts with the kernel run in edge-triggered mode. If there are no spurious notifications, the FD is re-armed to be polled by `epoll` or `kqueue` only if the status of the FD changes from "not ready" to "ready". Therefore, to make sure that the file descriptor is being polled before the fibre is blocked, an initial non-blocking `read` or `write` system call is required to re-arm the FD in the kernel and determine whether the fibre should be blocked at the user-level. When the notification arrives, the poller unblocks the fibre using the binary semaphore in the FD vector and places it back on a ready queue.

The overhead of issuing the next system call and blocking and unblocking the fibre for each connection that blocks can affect the performance of applications with a large number of connections. Moreover, In applications where the server and the client communicate back and forth over a connection, the server can expect the client to send a response or close the connection in the near future. Therefore, if the server can delay issuing the extra `read` system call, there is a possibility that the system call succeeds and the extra overhead mentioned above can be avoided altogether.

A low overhead solution in the runtime is that a fibre yields before executing the `read` system call, and executes the system call after it runs again. The fibre context switches back to the ready queue of the same processor, and depending on the load on that processor, it takes a while before it executes again. This added delay before calling the `read` system call might be enough for the client to send the next request and hence the connection becomes ready for `read`. However, if the `read` system call fails, the overhead of context switching is added on top of the overhead discussed above. Then again the overhead of context switching is very small in comparison with issuing a system call.

This mechanism is evaluated in Section 4.3 and the results show a significant improvement in performance when fibres yield before issuing the `read` system call. Hence, by default the runtime uses this approach internally for all input I/O functions, but it is possible to disable it at compile time. Another side effect of this scheme is improved fairness. In the absence of preemption, yield before read reduces the possibility of a long-running fibre to starve others.

Batching

Fibres are blocked on an FD using a user-level semaphore associated with the FD in the FD vector and wait for the poller to notify them about the readiness. The poller calls the

epoll_wait() system call to determine the readiness of FDs. When an FD becomes ready, the poller simply unblocks the fibre waiting for that FD using the associated semaphore for input or output. When multiple FDs are ready, the unblocked fibres are by default placed back on the ready queue of their respective processors. However, placing the unblocked fibres back on each processors' queue one by one can introduce some overhead due to the synchronization required for each operation. Therefore, the runtime provides an optional batching of unblocking fibres, where fibres are first placed in a local unprotected queue, and then are transferred to each processor's queue in batches.

Lazy Registration

When a connection is short and only contains a small request to be processed, after the connection is established, the client sends the request, the server processes the request and sends back a response, and finally the connection is closed. Usually, when a connection is established, the request from the client is ready to be consumed by the server. Also, writing to a socket is less likely to block unless the kernel-buffers are full. Therefore, in scenarios where a connection is short, there is a good chance that none of the system calls block and hence registering and unregistering the FD with the kernel and the user-level synchronization facility only adds overhead without being used.

Accordingly, the runtime provides an optional *lazy registration* of FDs. Using lazy registration, the registration of a FD is postponed until the connection blocks. Hence, the system call overhead for registering and unregistering the FD with `epoll` and `kqueue` is avoided if the connection does not block. Lazy registration is evaluated against always registering FDs in Section 4.3 and the results show that lazy registration can significantly improve the performance of applications with short connections. Hence, the runtime by default enables lazy registration but provides a compile time flag to disable it.

3.4.3 Polling Mechanism

As explained in the previous section, when a fibre is blocked on a file descriptor, an asynchronous polling mechanism is needed to frequently poll the network and unblock the fibre when the connection becomes ready. The asynchronous nature of the polling mechanism faces two design challenges: 1) how does the runtime poll the network? 2) how frequently should the network be polled?

There are two possible approaches to implement a polling mechanism. First, a distributed polling scheme where all worker processors in a cluster can poll the network. The

second approach is using a central polling mechanism through a separate kernel-thread, i.e., a *poller thread*. This thread polls the network in a loop and the worker processors are not involved in polling. The runtime supports both distributed and central polling mechanisms.

It is important to note that each cluster has a separate `epoll` or `kqueue` instance, and only a single kernel-thread (or processor) per cluster is allowed to poll at a time. The runtime makes sure that only a single thread is polling the network and accessing the polling instance. Since only a single thread at a time is polling, this approach does not introduce the thundering herd problem.

Poller Fibre

Distributed polling can be done in two ways; either polling directly by worker processors, or using a *poller fibre* along with the facilities provided by the scheduler to schedule and execute the poller fibre. The distributed polling design is provided in both Go and μC++. The former encapsulates the mechanism directly in the runtime and the latter uses an extra user-level thread to poll the network. Since using a poller fibre is simpler and provides more flexibility, the runtime supports the latter and processors poll through a poller fibre. Therefore, an extra fibre per cluster, i.e., poller fibre, is used to perform the polling and this fibre can be executed by any of the processors in a cluster.

The poller fibre is by default placed in the background queue of a cluster, i.e., processors only execute the poller fibre before going idle. There is only a single poller fibre per cluster, and therefore only a single processor at a time can poll the network, even if more than one processor is becoming idle. However, if processors perform a blocking poll and there are no notifications to deliver, the blocking call can block for a long time. The main task of a worker processor is to execute fibres, but blocking a processor for a long time prevents it from taking part in the execution of fibres. One solution is to switch back and forth between polling and checking the ready queues using a timer. However, this design can become very complex and requires adjusting the timers for different workloads.

As an alternative solution, since both `epoll` and `kqueue` can be applied hierarchically and the poll FDs can also be polled, a global master poller thread is used to poll the status of the poll FDs. The polling fibre is only scheduled to run by the master poller when there are notifications to deliver and the complexities of handling timers are avoided. Hence, when a processor executes the poller fibre, it performs a non-blocking poll. However, the master poller performs a blocking poll waiting for the poll FDs to become ready. This design is

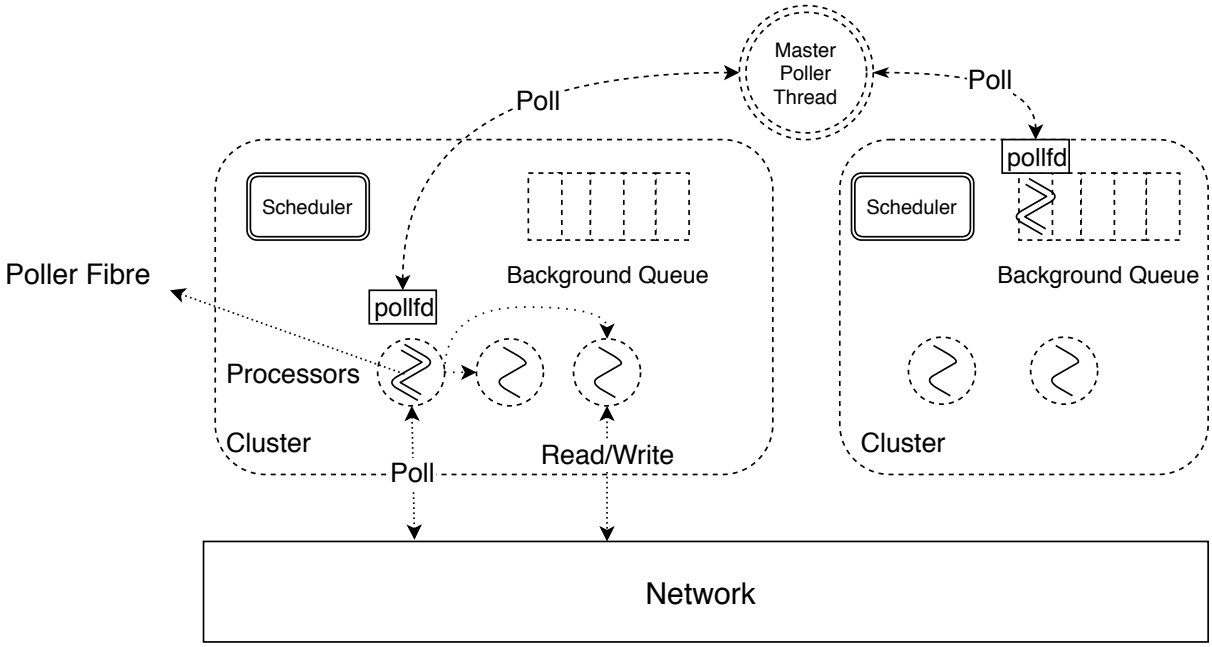


Figure 3.7: Design with a poller fibre per cluster.

illustrated in Figure 3.7. This design does not deliver poll notifications with the lowest possible latency, but typically provides sufficient throughput, as shown in Section 4.3.

Poller Thread

The alternative to distributed polling is to poll using a single dedicated *poller thread* that performs blocking polls in a loop. Therefore, the thread is blocked as long as there are no notifications to deliver. But as soon as a notification arrives, the thread becomes active and unblocks fibres that are waiting on the ready FD. This design is illustrated in Figure 3.8.

However, the frequency at which the poller thread polls the network can affect the performance of the application. The system calls that are used to poll have non-negligible overhead, so the frequency of polling presents a trade-off between overhead and latency. Polling very frequently can add overhead and be inefficient, but polling rarely can lead to long tail latencies. This trade-off is aggravated by spurious poll notifications (explained below). Figure 3.9 illustrates the polling frequency spectrum. On one end of the spectrum the poller polls the network in a loop, and on the other end it does not poll the network

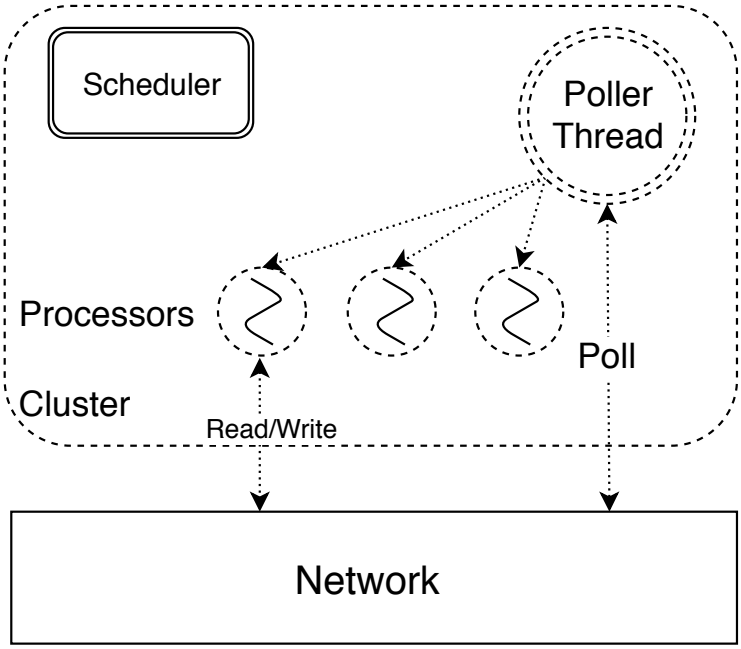


Figure 3.8: Central polling mechanism using a dedicated poller thread.

at all.

Four design alternatives are supported by the runtime that cover different parts of the frequency spectrum:

- *loop*: In this variation, the poller thread simply calls the blocking poll system call in a loop. Therefore, as soon as the poller unblocks the blocked fibres, another poll system call is issued. Hence, the runtime is always polling the network.
- *loop-sleep*: This design uses a polling loop but lowers the polling frequency by forcing the poller thread to go to sleep after each poll. The longer the sleep time, the lower the polling frequency. Not sleeping at all is equivalent to the previous version and always sleeping means no polling is performed at all as illustrated in Figure 3.9.
- *opportunistic*: Another design is poll only if necessary, i.e., when at least one of the worker processors is idle. Therefore, the poller thread is only executed if at least one of the worker processors runs out of work. Hence, the polling frequency is tied to the workload and the status of the processors and it sits somewhere in the middle of the spectrum.

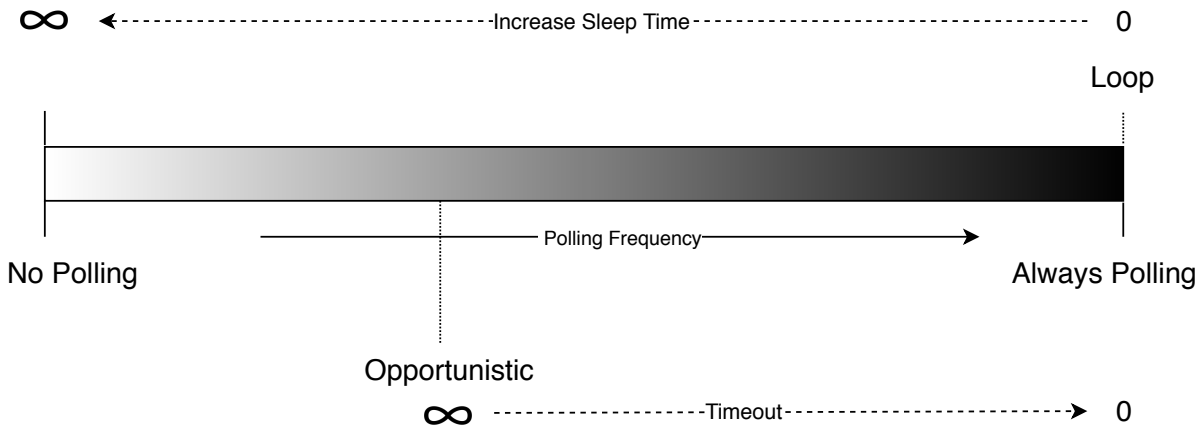


Figure 3.9: Polling frequency spectrum.

- *opportunistic-timeout*: Since the polling frequency of the previous version is tied to the characteristics of the workload, a very busy workload can have large polling intervals. This delay can affect the latency of the connections that are blocked and are not being polled because the poller thread is not triggered. A solution to address this problem is to use a timeout to increase the polling frequency. If the poller is not triggered before the timeout expires, the poller thread is activated and polls the network. Setting the timeout to 0 makes the design equivalent of the *loop* version.

These design alternatives are evaluated in Section 4.3 and their effect on throughput and latency is studied in more detail.

3.5 Implementation Details

Each fibre has a stack that automatically captures the state of the fibre and is used by the runtime to suspend and resume that fibre. Fibres operate like a traditional thread but context switches happen at the user level as explained in Chapter 2.

Moreover, each fibre can have a fixed or dynamic stack size, which is configurable at compile time. When fibres have a fixed-size stack, the size of the stack can be adjusted when the fibre object is being created. The stack memory is allocated when the fibre object is created and destroyed when the destructor of the fibre object is called. A protection

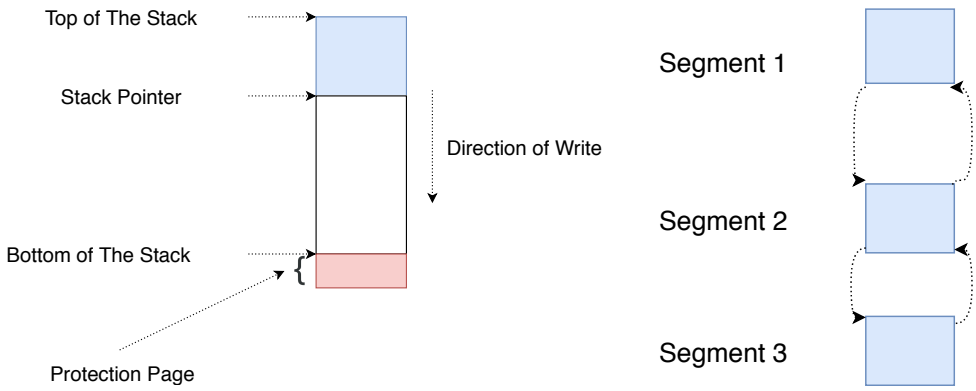


Figure 3.10: (a) Fixed-size stack, (b) Segmented stack

page is used to detect stack overflows. This page is the last page at the bottom of the stack and writing to this area or reading from this area results in termination of the application. The protection page is created using *mprotect* system call which can add some overhead when fibres are created. Figure 3.10a shows the layout of the static stack in memory.

In addition, the runtime provides segmented stacks using the split stack feature supported by both GCC and Clang [200]. Split stack provides a fast mechanism to check the stack pointer against the bottom of the current stack at each function call. This mechanism relies on an extension of the fibre context switch to also switch the dedicated split-stack context. Segmented stacks are lightweight and only add a few instructions to each subroutine. Figure 3.10b shows the layout of the segmented stacks in memory.

Fibres are scheduled cooperatively, i.e., each fibre runs until it voluntarily yields to other fibres, migrates to another cluster, blocks on an external event, or reaches the end of its execution. Stack contexts are suspended and restored by pushing and popping the callee-owned registers to and from the stack. Currently, the runtime only supports x86-64 architectures. The runtime does not use *ucontext_t* in Linux or FreeBSD, and uses custom assembly code to perform the context switches.

Fibres also provide *join* and *detach* semantics. If *join()* is called on a fibre, the calling fibre blocks until the callee fibre is done and unblocks the caller. Also, fibres can be detached, which means the caller fibres that try to join with this fibre are not blocked. Both fibre and processor objects support join-on-destructor safety semantics, i.e., deleting an object is automatically deferred until the respective execution contexts have run to completion.

The runtime provides a range of intrusive data structures that are used to avoid extra memory allocations when moving fibres around and for additional safety. For instance all blocking and ready queues in the runtime are implemented using intrusive linked-lists (including lock-free variations). These data structures are also used to manage processors in a cluster. For instance, they are used to implement busy and idle lists for each cluster.

Two variations of blocking lock-free Multiple Producer Single Consumer (MPSC) queues are implemented: The Nemesis queue [40] and a *stub queue* inspired by [217]. Both queues are lock-free, but not wait-free; the consumer might have to wait for changes from producer to be finalized, but this window is very small. The Nemesis queue is not modified and the original algorithm is used for implementation. However, the original implementation of the stub queue is not fully intrusive, and therefore is modified to be fully intrusive and meet the requirements of the runtime. Also, the original implementation requires an additional copy operation when an item is removed, which is fixed in the modified version used in the runtime.

Applications might need to interact with the runtime and access information about the currently running fibre, processor, and the cluster domain. Therefore, each processor provides a low overhead mechanism through a set of *thread-local* pointers to these objects.

The processors in a cluster can be confined to only execute on a set of cores. This functionality is not provided by the runtime and is left to the application developer to make that decision according to the requirements of the application. For instance, all the processors in a cluster can be pinned to a set of cores in a socket that has a local NUMA node. Therefore, the fibres that belong to the cluster can only execute on these cores and hence avoid the cost of remote NUMA node accesses. The runtime provides direct access to the underlying kernel thread. Therefore, if required, applications can access the thread and set the affinity at the application level. This provides more flexibility and control for the application developer.

In addition, there is limited, proof-of-concept support for debugging by using GDB's extension API for Python. With this extension, it is possible to view the execution context of each fibre and perform backtraces per fibre in addition to the default support for kernel threads provided by GDB.

Chapter 4

Evaluation

In this chapter, efficiency and scalability of I/O multiplexing, synchronization, and scheduling components presented in Chapter 3 are evaluated and compared with other runtimes. First, various design alternatives for each component presented in Chapter 3 are evaluated individually and next, the runtime is evaluated against other runtimes. Also, Memcached is used to evaluate the runtime and thus all these components together using different workloads.

4.1 Environment and Tools

All experiments are performed on a 4-socket, 64-core AMD Opteron 6380 with 512 GB RAM, but have also been repeated on a 4-socket, 32-core, 2xHT Intel Xeon E5-4610v2 with 256 GB RAM. The respective cache layout of both machines is shown in Table 4.1. Some experiments on the Intel platform are scaled down to avoid using hyper-threaded execution. However, the results are generally consistent on both platforms. Only the AMD results are presented in this chapter, because of the higher core count that is possible on this machine. The results from the Intel machine are presented in Appendix B.

The operating system on both machines is Ubuntu Linux 18.04, updated with recent packages, and running kernel version 4.15.0-22-generic. For all experiments, the specified number of cores is provisioned using the `taskset` utility. All data points show an average of 20 runs. The maximum Coefficient of Variation (CV) for each experiment is reported, but do not plot error bars, because the CVs are small and error bars would only distort the graphs.

The efficiency of various design alternatives for the I/O subsystem is evaluated using micro-benchmarks, a simple web server, and Memcached. Micro-benchmarks are used to investigate the fundamental building blocks of user-level runtime systems. The simple web server is used to evaluate the I/O subsystem by serving connections from a variation of the *plaintext* benchmark from the TechEmpower web framework benchmark suite [201]. A web server receives simple and short Hypertext Transfer Protocol (HTTP) requests from clients and responds to each request with a plain "Hello, World!" HTTP response. Memcached is an event-based key-value storage that is used extensively as an in-memory cache.

The 'wrk' [77] and 'weighttp' [125] benchmark programs are used to generate load for the web server experiments. Using 'wrk' each connection sends a request and waits for the response from the server in a closed loop. 'weighttp' only sends a single request per connection and closes the connection when the response is received. Mutilate [119] is a close-loop load generator for Memcached and is used to evaluate Memcached. Memcached and web server experiments are performed with sufficient concurrency. However, since the clients are running on a single machine, pipelining is used to send multiple requests over a single connection to the server to represent a larger number of concurrent clients. Pipelining allows the client to send multiple requests back-to-back without waiting for the server to respond to previous requests.

Moreover, Memcached and web server benchmarks are executed locally on a multi-core computer partitioned to run both client and server to eliminate the cost of device communication. Hence, this experiment represents a worst-case scenario for a user-level runtime by exposing the overhead from I/O multiplexing without any mitigating factors – to the extent possible. Only M:N user-level runtimes are included in this study and the best-in-class event-driven system, ULib and Memcached, are compared against the runtime. ULib [183], is an event-driven web server that is among the top performer in TechEmpower benchmarks [201].

	level	type	size	shared
AMD	L1	instruction	64 KB	2 cores
AMD	L1	data	16 KB	no
AMD	L2	unified	2 MB	2 cores
AMD	L3	unified	6 MB	8 cores
Intel	L1	instruction	32 KB	(2 HTs)
Intel	L1	data	32 KB	(2 HTs)
Intel	L2	unified	256 KB	(2 HTs)
Intel	L3	unified	16 MB	8 cores

Table 4.1: Cache Layout

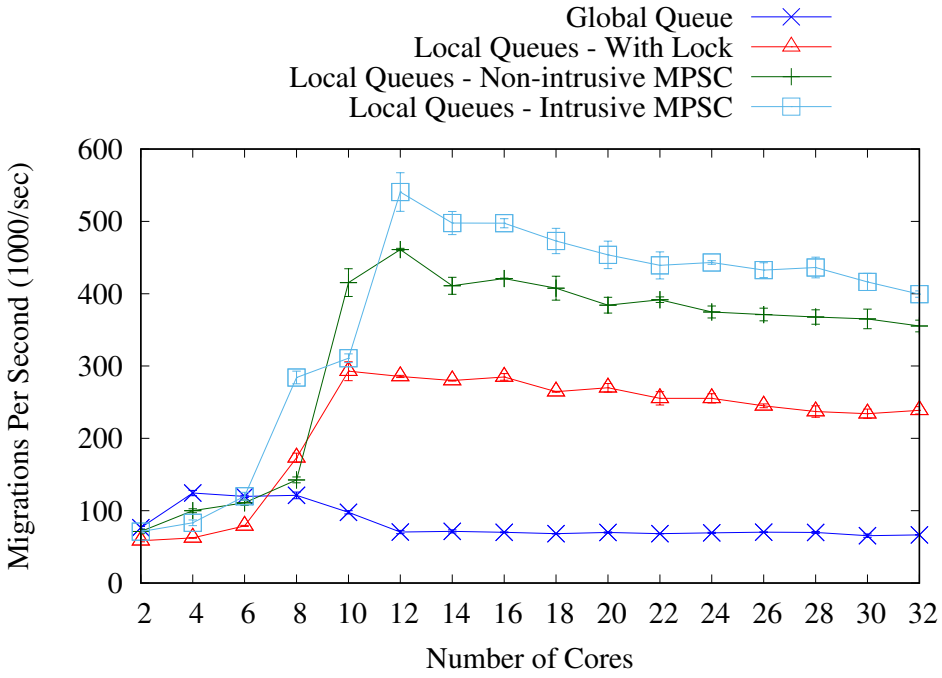


Figure 4.1: Scalability of Different Queues

4.2 Queue Performance

As discussed in Chapter 2, the scheduler can use work-sharing or work-pushing strategies for initial placement of fibres. The first experiment evaluates the scalability of each strategy

through a benchmark. The experiment covers work-sharing with a single shared queue and three variations of work-pushing using different local queue designs. The benchmark creates two clusters each containing the same number of processors and each processor creates 1 million fibres. Every fibre migrates 10 times back and forth between the clusters, and after that it exits. This benchmark is designed to stress test each queue and therefore, fibres do not perform additional computation and as soon as a fibre is scheduled it migrates to another cluster. In reality, fibres perform some computation, hence, processors access queues not as frequent as during the benchmark.

The results are illustrated in Figure 4.1. The number of cores is incremented by two since adding a single processor to each cluster results in utilizing two more cores. The results confirm that work-pushing with a local queue per processor has better scalability than work-sharing with a single shared queue in a cluster. Among the variations with local queues, the intrusive MPSC variation provide higher performance across different number of cores and the non-intrusive variation follows behind. The variation with a single lock protecting the local queue has the lowest performance among the designs with local queues. Therefore, the intrusive MPSC queue along with work-pushing is used as the default strategy to provide good scalability across different number of cores in the runtime.

4.3 I/O Subsystem Design

Multiple design alternatives for the I/O subsystem are presented in Chapter 3. All these alternatives are evaluated in this section and these results are used to determine the default behaviour of the I/O subsystem.

4.3.1 Yield Before Read

Fibres yielding before calling the `read` system call delays reading from the socket and can improve the rate of successful `read` system calls in busy systems. Consequently, the overhead of unsuccessful system calls and blocking and unblocking the fibre is also avoided. Figure 4.2 illustrates the effect of this design on the performance of a web server. The experiment uses HTTP pipeline depth of 1 and 4 across different number of cores using 4,096 connections. For this experiment the default polling mechanism is used.

The results show that yielding before `read` increases the throughput across different parallelism levels. Also, the difference between both versions increases with larger number of cores. Increasing the number of cores increases the capacity of the system and therefore

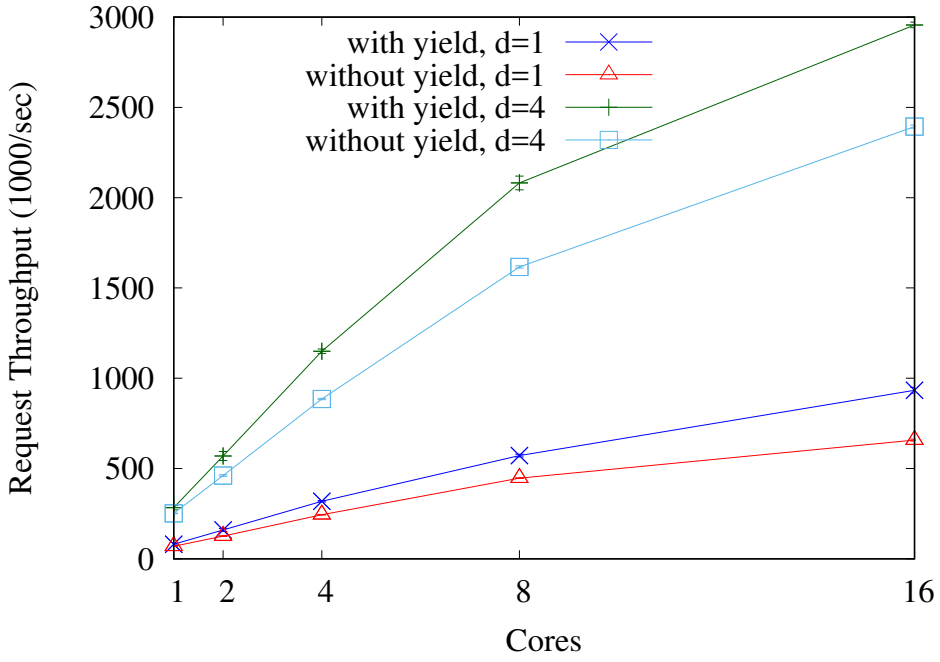


Figure 4.2: Effect of yielding before read on a web server performance.

the cumulative overhead of unsuccessful read system calls increases. But yielding reduces this overhead by decreasing the number of unsuccessful reads. Since yielding improves the performance, this feature is enabled by default but can be disabled at compile time.

4.3.2 Lazy Registration

In Chapter 3 lazy registration is identified as a method of avoiding the overhead of registering and unregistering the file descriptors when workloads include short connections. Figure 4.3 illustrates the effect of adding lazy registrations on the throughput of the web server. The web server is used to serve short connections with and without lazy registration. Weighttp [125] benchmarking tool is used to generate short connections where each connection only contains a single HTTP request and it is closed after the response is received. According to the results, lazy registration improves the throughput by almost 60%. This experiment demonstrates that avoiding the overhead of unnecessary system calls and notifications can significantly improve the performance. Therefore, by default the runtime supports lazy registration and only registers a FD to be polled when one of the system

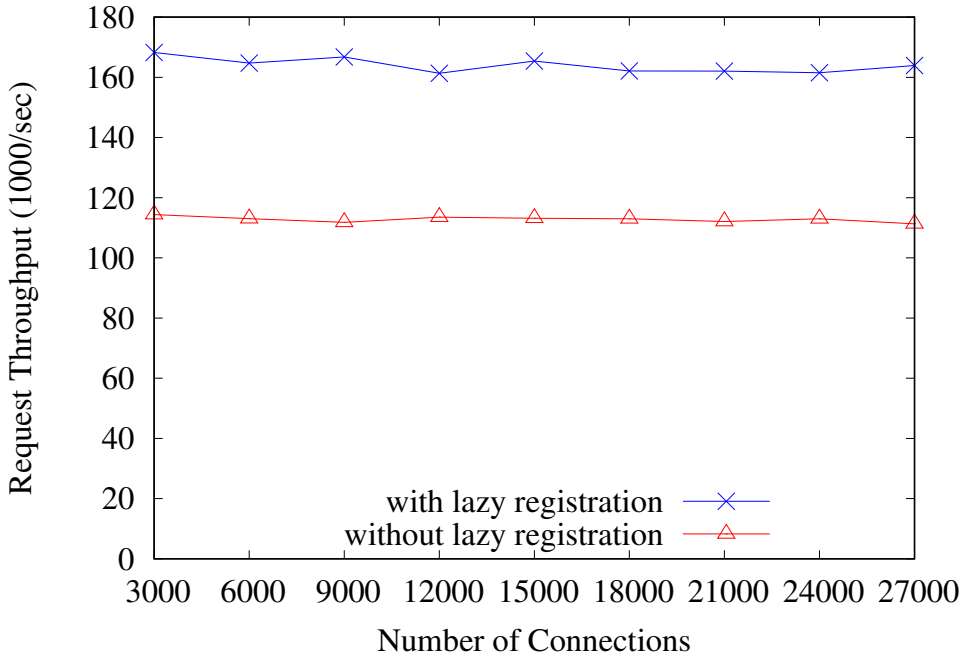


Figure 4.3: Effect of lazy registration on short-lived connections using a web server.

calls is not successful.

4.4 I/O Polling

The runtime supports multiple I/O polling mechanisms as explained in Chapter 3. A *polling fibre* can be used for distributed polling and a *poller thread* can be used for polling with a dedicated thread. Moreover, different polling frequencies can be used for the latter by choosing one of *loop, loop-sleep, opportunistic*, or *opportunistic-timeout* designs. The modified version of Memcached is used to evaluate the performance of each design variation and compare them against each other and against the original event-based version – termed 'Vanilla' for this presentation. There are 2048 concurrent connections and the depth of the pipeline is changed. The depth of the pipeline represents the number of simultaneous GET and SET operations sent by the client to the server. Figure 4.4 illustrates the throughput and read latency for the experiment.

The *opportunistic* and *opportunistic-timeout* variations are showing the lowest throughput and highest latency among all the versions across different pipeline depths. Both versions only poll the network when at least one worker processor runs out of work but the *opportunistic-timeout* version also polls the network every 10 ms. Hence, when all processors are busy for a long time the network is not going to be polled during that period. This delay can lead to unfair service time for serving the connections as some of the fibres that have their connection ready cannot execute while a few fibres that do not block keep executing. This scenario also results in longer average latencies, more variations in latencies, and also heavier latency tails as illustrated by the results.

The *opportunistic-timeout* variation with 10 milliseconds timeout delivers low throughput when the pipeline depth is small, but the performance improves as the pipeline depth increases. Investigating further reveals that when a processor runs out of work it tries to steal from other processors before signalling the poller thread, and the work-stealing adds delay and indirectly affects the polling frequency. Also, the additional time spent to find work replaces useful cycles that could be spent serving the blocked connections. Although the 10ms timeout helps to avoid the very long latencies in the *opportunistic* version, it does not completely remove the long latency tail.

Changing the timeout to 1 millisecond improves the throughput and overall latency. The maximum polling interval is now set to 1ms and is independent of the status of the worker processors. Therefore, blocked fibres are unblocked more frequently without relying on the status of the worker processors. Also, the processors work-stealing stage is not reach as frequently as with 10ms timeout, since blocked fibres are unblocked more frequently and pushed to the local queues of the processors by the poller thread.

The *loop* variation, where the poller thread performs blocking poll calls in a tight

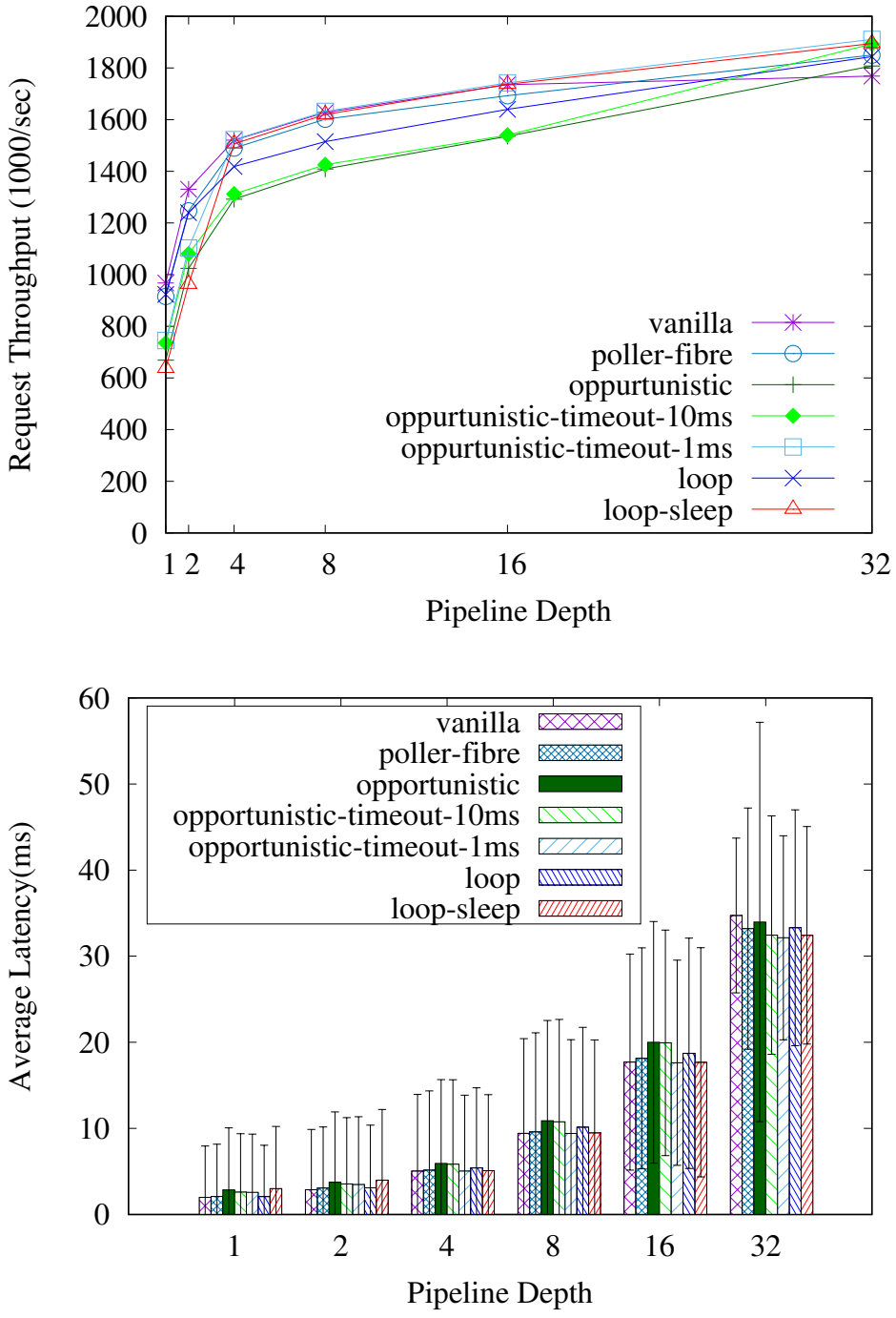


Figure 4.4: Throughput and Latency of I/O polling mechanisms.

loop, has overall lower throughput than *loop-sleep* and *opportunistic-timeout-1ms*, but has higher throughput than *opportunistic* and *opportunistic-timeout-10ms* variations. Polling in a loop without any sleeps means that the network is polled very frequently. Hence, FDs are not unblocked in batches and as soon as a single notification is available the call to the poll returns. Therefore, most fibres are unblocked one by one or in small numbers, which leads to more system calls to deliver all notifications in comparison to unblocking fibres in batches. Moreover, the poller thread is active most of the time, and this thread competes with worker processors for computing time. This leads to more frequent kernel preemptions of the worker processors and therefore introduces some overhead.

To mitigate this problem, the *loop-sleep* variation introduces a 1 millisecond sleep after each blocking poll. Hence, the polling frequency is reduced in comparison to the *loop* version, which leads to lower polling overhead. A larger polling interval removes frequent preemptions, and also leads to delivering the notifications in batches. This variation provides a better throughput than the *loop* version.

Both *loop-sleep* with 1ms sleep and *opportunistic-timeout* with 1ms timeout show comparable performance to the Vanilla version. The *loop-sleep* version has a fixed polling interval of 1ms regardless of the status of the processors. This configuration can become problematic for workloads with low concurrency where processors run out of work frequently. In this situation, the processor might have to block and wait for 1ms before more fibres are unblocked. In contrast, the *opportunistic-timeout* version has a *maximum* 1 ms polling intervals. Since processors signal the poller thread when they run out of work, the problem with low concurrency workloads does not become an issue.

The *poller-fibre* variation shows high throughput and low average latency starting at pipeline depth 4. The *loop-sleep* and *opportunistic-timeout* variations with 1ms sleep/-timeout along with the *poller-fibre* variation show better overall performance than other variations across various pipeline depths. This is not true for the pipeline depth of 1 and 2 where the *loop* variation has the highest throughput among other variations. Hence, the polling mechanism depends on the characteristics of the workload. However, to simplify the runtime, the tight loop variation is used as the default polling mechanism.

The event-based version has lower throughput and higher latency than all the versions of user-level threading with pipeline depth 32. This is due to the extra overhead imposed by having a polling instance per thread which leads to larger number of system calls to handle this workload in comparison with the fibre runtime.

4.5 Benchmark Evaluation

In this section, the synchronization and scheduling components of the runtime are evaluated against other user-level threading runtimes. The benchmark represents a simple application with L locks, T threads and C cores. Each thread first executes a non-critical section part of code for t_{ncs}. After a thread acquires the lock it executes the critical section and runs for t_{cs} and then releases the lock. After releasing the lock, the thread goes back to executing the non-critical section of the code. The critical and non-critical sections are implemented with a work loop that keeps the CPU busy for t_{cs} and t_{ncs} accordingly. The assumption is all critical sections have the same computation time. Algorithm 1 provides the pseudo code for the benchmark.

Algorithm 1 Benchmark Pseudo Code

```
1: C: Number of cores
2: T: Number of threads
3: L: Number of locks
4: t_cs: Duration of each critical section work loop
5: t_ncs: Duration of each non-critical section work loop
6: procedure ThreadFunction()
7:     while true do
8:         Run Work-loop For t_ncs
9:         l_r = Random Lock
10:        Acquire l_r
11:        Run Work-loop For t_cs
12:        Release l_r
13: For each thread call THREADFUNCTION()
```

Both locked and unlocked work loops can be calibrated to a specific time period. For simplicity, the experiments reported here use symmetric settings for unrestricted and critical-section work. Other runtime parameters are explained as needed. For this particular benchmark with a tight loop, the set of cores for the AMD platform is restricted to one of each two-core package (cf. Table 4.1), for a total of 32 cores, to avoid low-level caching effects that might arise from other particular placements of threads on cores.

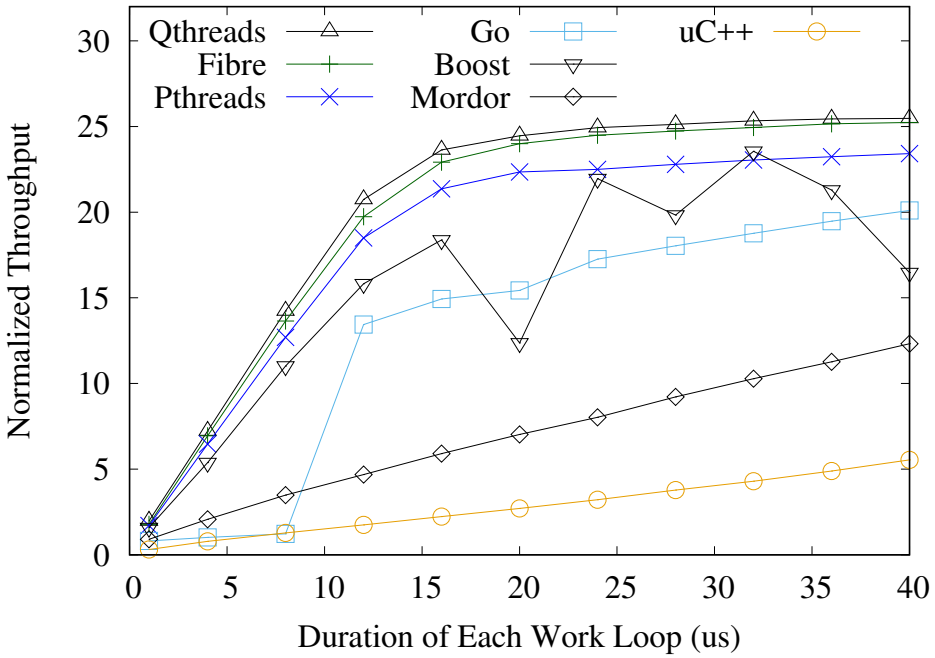


Figure 4.5: Scalability (32 cores, 1024 fibres, 32 locks). Throughput is normalized by the respective throughput for a single thread/fibre.

4.5.1 Scalability

Most multi-threading runtime systems perform well when the amount of work between synchronization and scheduling points is large. This experiment is designed to investigate scalability by determining the amount of work that is necessary for various runtime systems to provide good scalability at non-trivial core and thread counts. The benchmark has 32 cores available and executes 1024 threads or fibres. It is configured with 32 locks, which cause some, but not excessive, contention and synchronization overhead. The work loop duration is varied. Shorter durations are more affected by locking and scheduling overhead, and as the duration is increased this overhead should become insignificant. Therefore, the scalability of the runtime can be measured by increasing the work loop duration, as longer durations reveals the overhead that the runtime imposes on the experiment. Throughput is normalized by the respective throughput for a single thread/fibre. The results are shown in Figure 4.5 and B.1. The runtime system presented in this thesis is labelled 'Fibre', while 'Boost' refers to the Boost Fibre implementation. 'Pthreads' means directly using system threads as application threads.

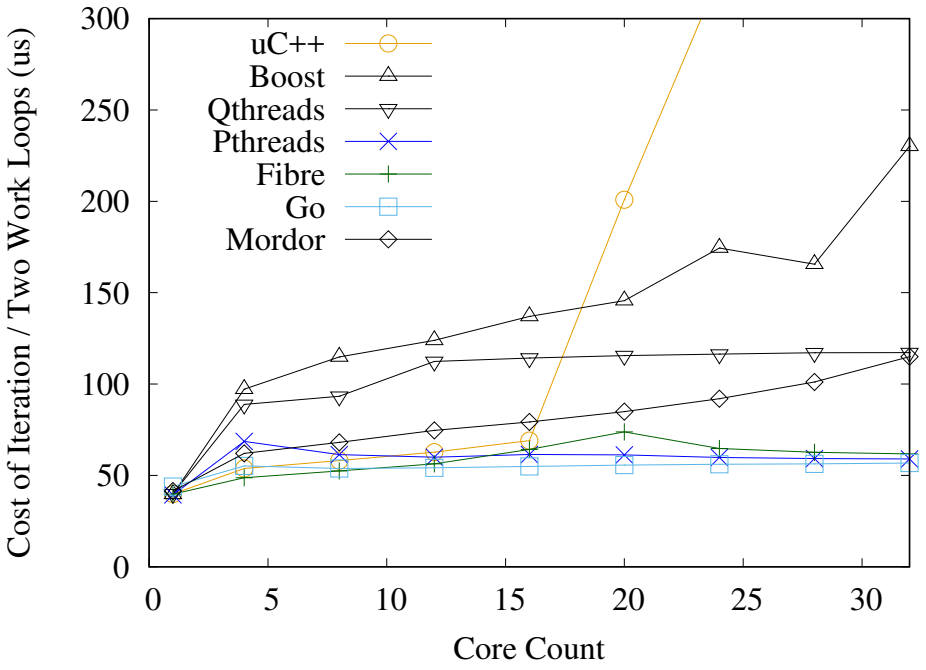


Figure 4.6: Efficiency (N^2 fibres, $N/4$ locks, 20μs loops), lower is better

Four of the runtime systems show very similar scalability in this experiment: Qthreads, Fibre, Pthreads, and Go. Occasional experiment runs for Boost perform very poorly, which reduces the average for the corresponding data points and distorts the curve. Manual inspection shows that the best possible scalability of Boost would be slightly lower than the top four runtimes. The scalability of Mordor is generally lower than the previously mentioned runtime systems. μC++ has the poorest scalability of the tested runtimes. It is caused by using a scheduler with a central ready queue, as discussed in Section 3.2. Additional experiments (not shown here) confirm that μC++'s scaling curve starts rising at 55 - 70 μs for both work loops, beyond which it shows scalability close to the top runtimes.

4.5.2 Efficiency

The next experiment investigates the runtime overhead for locking and scheduling. The previous experiment indicates that most runtime systems show reasonable scalability when both work loops are set to 20μs. Therefore, that setting is used in this experiment. The

number of cores is varied and the number of fibres is set to the square of cores. The runtime overhead is measured using the Linux cgroups' `cpuacct` subsystem and divided by the number of observed work iterations. The overhead is reported as the cost of each work iteration, which is comprised of one unrestricted work loop and one critical-section work loop, plus synchronization overhead. Thus, any value in excess of $2 \cdot 20 = 40\mu$s represents runtime overhead. Since overhead becomes more visible with higher contention, the number of locks is set to $N/4$ for N cores, which ensures a sufficient level of contention.

The results are shown in Figure 4.6 and B.2. μC++ makes heavy use of spin locks internally, which probably causes its limited scalability to translate into poor efficiency beyond 16 cores in this experiment. Both Boost and Qthreads show good scalability in the previous experiment, but sub-optimal efficiency in this one. The explanation is that both the Boost and the Qthreads runtime system do not fully support blocking synchronization. While individual fibres might be blocked from execution, these runtime systems never suspend the underlying system threads. Instead, all available threads spin continuously while seeking work. This approach is a suitable strategy for HPC applications that usually keep a dedicated set of cores busy. However, it is not an acceptable scheduling regime for general-purpose multi-threading applications that operate in a shared computing environment, especially when also considering power conservation. Mordor shows good efficiency when the number of cores are small, but it is not very efficient as the number of cores increases. Pthreads, Fibre, and Go provide effective blocking and suspension and show similar efficiency.

4.5.3 I/O Multiplexing

The web server benchmark is used to evaluate the performance of the I/O subsystem and comparing it against other runtimes. Both Boost Fibres and Qthreads do not support fibre blocking and system thread suspension, and do not provide an I/O subsystem comparable to other runtime systems. Mordor does not show competitive scalability in the basic experiments, and it has not been possible to develop an efficient web server implementation, due to the lack of publicly available documentation. Thus, none of the above are considered for further evaluation. However, ULib is added as the best-in-class event-based system for comparison.

The experiments compare the ULib web server with a benchmark web server using Go with the Fasthttp [210] package (derived from the TechEmpower benchmark code) and a minimal multi-threaded web server that can use either POSIX threads, the fibre runtime, or μC++. The ULib web server uses a sophisticated and highly-tuned N-copy architecture

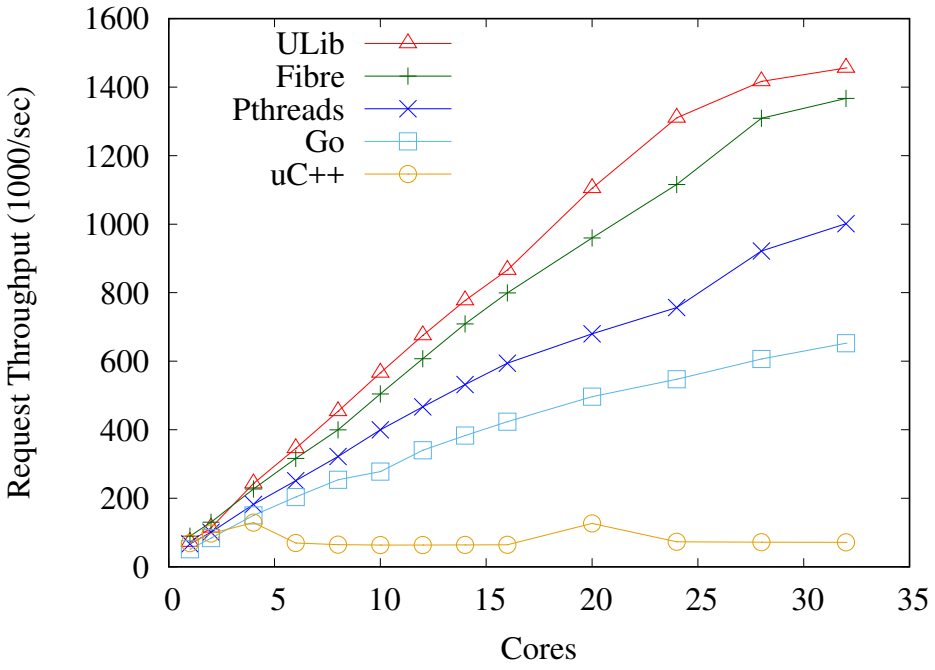


Figure 4.7: Web Server (12,000 connections)

that does not permit any load balancing or migration after connection establishment. For example, the server socket is cloned across all processes using the `SO_REUSEPORT` socket option. In addition, to exploit multiple cores, ULib clones multiple processes that do not share file descriptors with each other which makes it impossible to do any form of load balancing. In contrast, the Go server and the test server use a simple and straightforward architecture with a dedicated 'acceptor' thread launching per-connection worker threads for incoming connections. For both the fibre runtime and μC++, the cluster mechanism (cf. Section 3.2) is used to limit a scheduling and I/O domain to 16 cores. Thus, experiments with more than 16 cores for the server use multiple scheduling clusters. All servers, including ULib's server, benefit from a warm-up period before the actual performance measurements. The maximum CV observed during these experiments is 0.08.

The 'wrk' benchmark program [77] creates 12,000 connections and then exchanges requests and responses as fast as possible, using 32 out of the 64 cores, for 30 seconds. Multiple connections provide a means for latency-masking concurrency in each client thread and the experiment does not use additional HTTP pipelining. The throughput is measured for a varying number of cores available to the web server. Beyond 28 server cores, the client

program becomes the bottleneck in the experiment. The results are shown in Figure 4.7.

μC++ scales well only up to 4 cores. The data points at 18 and 20 cores show how the second cluster helps scalability. Go scales reasonably well, but its throughput is significantly lower than other runtimes. Fibre performs almost as good as ULib in this stress test, while Pthreads shows an increasing performance gap, with an increasing number of cores, when compared to ULib and Fibre.

In a second experiment, reported in Figure 4.8, the number of cores is fixed at 16 and the number of connections is varied. In this experiment, the wrk client uses 48 cores to ensure that it does not become the bottleneck. The results show that all web servers and runtime systems can scale up to 60,000 concurrent connections. The relative performance is consistent with the observations from the previous experiment. Fibre is able to keep up with ULib, while Pthreads follows behind. Go has lower throughput than Pthreads, while μC++ has the lowest throughput in this experiment. It is especially remarkable that Pthreads can handle that many connections with 60,000 concurrently running system threads, albeit at a noticeable performance drop. The per-connection effective memory consumption for the threaded web server implementations were measured separately and range from 8 KB for Fibre to 12 KB for Pthreads.

The third experiment is similar to the previous experiment but the number of connections is increased up to 1 million as shown in Figure 4.9. This experiment only tests the limits of scalability of each runtime and is not meant to study the behaviour of the runtimes with large number of connections. All runtimes except Pthreads successfully handle up to 1 million concurrent connections (not all are handled successfully). Pthreads can handle up to 500,000 connections. Go has the lowest throughput among all the runtimes. The throughput of the fibre runtime is very close to ULib but the gap slightly increases as the number of connections increase. However, due to hardware limitations and the experiment methodology the results are tentative. Studying the behaviour of the runtime with very large number of connections is out of scope of this thesis and is left for future work.

The overarching conclusion drawn from these experiments is that all three runtime paradigms provide somewhat competitive I/O multiplexing performance. Event-driven execution slightly outperforms user threading, and both are noticeably more scalable than system threading. Of all user-level threading systems evaluated here, only the fibre runtime is efficient and scalable enough to perform on par with event-driven execution for these stress-testing I/O multiplexing benchmarks.

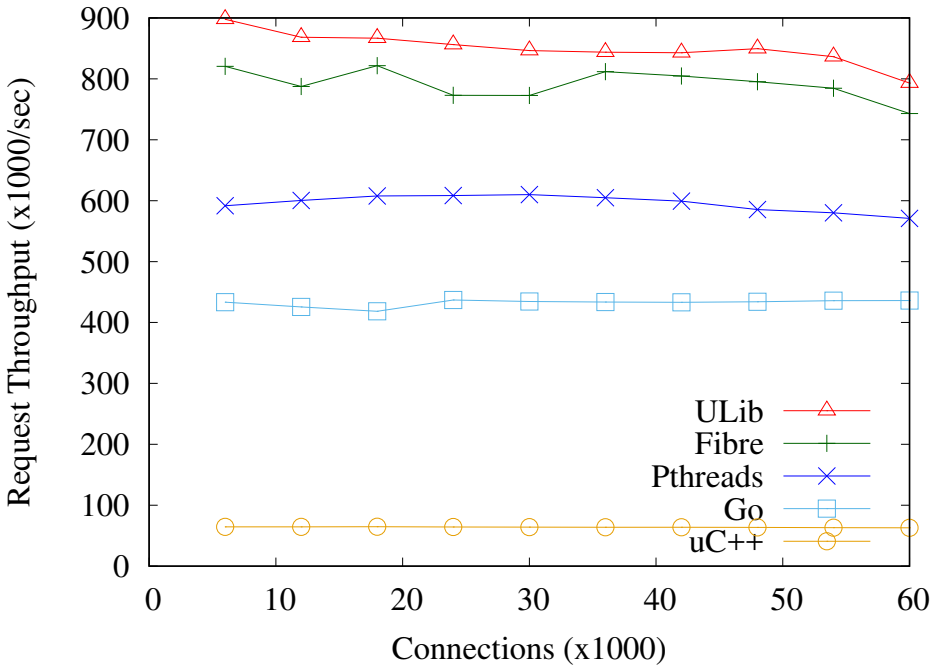


Figure 4.8: Web Server (16 cores)

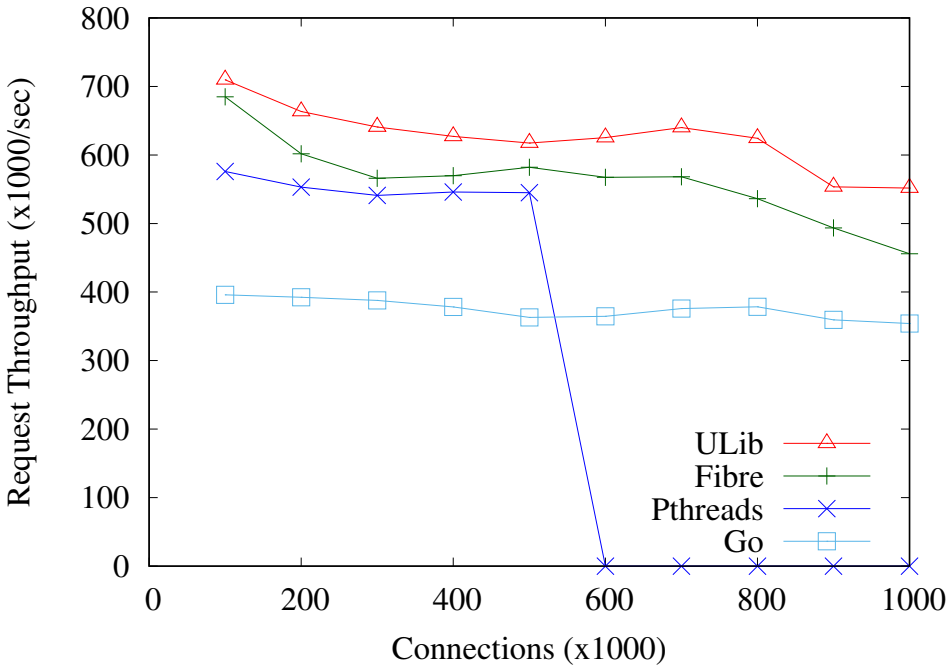


Figure 4.9: Web Server (16 cores)

4.6 Application Evaluation

Memcached is an in-memory key-value store, primarily used as an object cache for the results of dynamic queries in web applications. It is a mature software system that is deployed in a broad range of large-scale production environments [62]. The server is designed as an event-driven state machine, which is replicated across multiple system threads to utilize processor cores. It thus supports multi-core execution and shared mutable state, but has a simpler software structure than contemporary production-grade web servers or database servers. These characteristics make it the ideal candidate for studying different runtime paradigms.

An adaptation of the original Memcached source code is presented here that has a minimal set of modifications to convert the execution model to thread-per-connection without loosing essential functionality. In particular, the core state machine for processing requests is still present in the modified version, but it is synchronously driven by a separate thread or fibre for each connection. This approach enables a more equitable comparison of runtime systems than using two entirely different implementations of an application, but it is slightly biased in favour of event-driven execution. The minimal port of Memcached might not realize certain efficiency gains that would be possible with a completely synchronous multi-threaded implementation. However, given the state of the art in production-level systems, fine-grained multi-threading is clearly the challenger in this comparison. Therefore, an evaluation with a slight bias in favour of the event-driven model establishes a useful lower bound for the relative performance of multi-threading.

4.6.1 Memcached Transformation

Memcached provides a platform to test all the components of the runtime in a single application. It is a multithreaded network server that uses mutex locks to provide an in-memory key/value storage. Therefore, scheduler, I/O subsystem, and synchronization infrastructure can be evaluated together using Memcached. The original Memcached implemented an N-copy event-based programming model. Hence, to evaluate the user-level threading runtime, Memcached needs to be transformed to a fibre-per-connection model.

The first step to transforming Memcached for thread-per-connection processing is facilitated in 10 steps, each of which results in a working version with good performance. The combined changes of these modifications amount to adding 969 lines of code, while removing 373 lines and the dependency on libevent [133]. One more update replaces system threads with fibres resulting in another code delta of +554/-452 lines. In total,

this changes about 5% of Memcached's direct code base. These changes are applied to Memcached 1.5.7, which is the version used for the experimental evaluation. The updates remove some functionality and protocol operations that do not affect performance, but would increase the complexity of the port: per-thread statistics aggregation, application-level connection timeouts, as well as the monitoring commands 'lru_crawler metadump' and 'watch'. Also, Memcached uses stop-the-world synchronization to halt all operations during the resizing of its central hash table. The fibre runtime provides similar functionality, but incurs a somewhat higher latency. While this scenario does not matter in reality, since hash table resizing is extremely rare and never undone, it would affect short-term benchmark experiments. Therefore, this functionality is not used during the performance evaluation.

4.6.2 Performance Evaluation

The thread-per-connection version of Memcached is used to evaluate the performance of user-level fibres and system threads in comparison with the original event-driven implementation – termed 'Vanilla' for this presentation. Because the original structure and logic of the application is left intact, the evaluation can use simple and straightforward stress-testing experiment setups. The experiments use the Mutilate load generator [119]. In all experiments, the server uses up to 16 cores, while the remaining 48 cores (or 16 cores in the Intel machine) are used for load generation. This ensures that the load generator does not become a bottleneck and always provides ample work for the server. The Memcached server is always run with a sufficiently high memory limit (`-m 65536`) and connection limit (`-c 65536`) to accommodate all experiments. To eliminate the effects of hash table resizing from these experiments, the server is also started with a sufficiently sized hash table (`-o hashpower=28`). The load generator is run with a pipeline depth of 16 except where noted. It runs a 10-second warm-up period, followed by 60 seconds of experiment time. For all experiments, the average throughput, average CPU usage, and an averaged latency profile are reported. The latency profile shows the 10th percentile, average, and 90th percentile request latency as measured by the client, each averaged over the experiment runs. Except where noted, the maximum CV is 0.05.

Different Runtime Systems

The first experiment investigates scalability with an increasing number of cores. It is set up with 6,000 connections. The results can be found in Figure 4.10. All systems perform similarly in terms of throughput and scale well up to 16 cores. As expected, the CPU usage

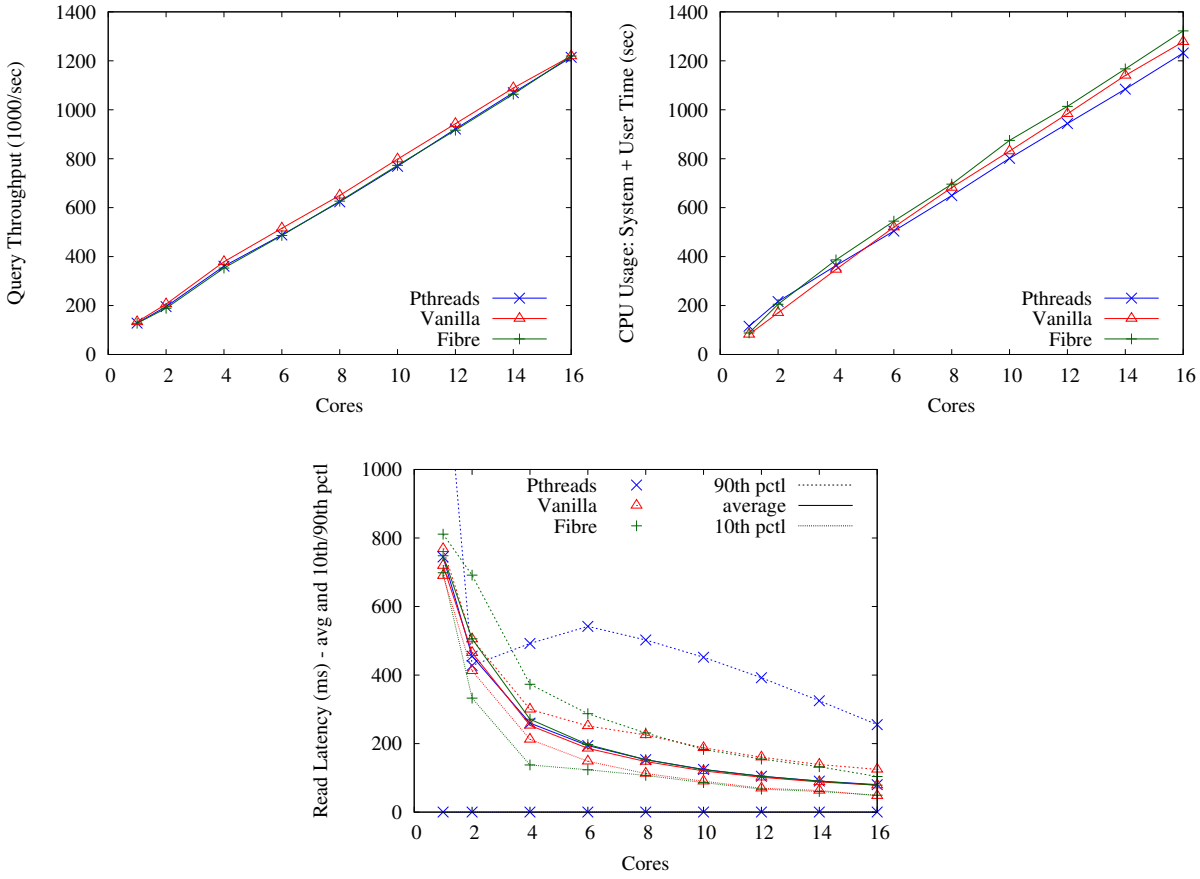


Figure 4.10: Memcached - Core Scalability (6,000 connections, write ratio 0.1)

increases linearly. The latency curves for Vanilla and Fibre also look as expected. However, for the Pthreads implementation, while the average latency is similar to the other systems, the 10th percentile is always extremely small, while the average 90th percentile latency is quite long and even crosses the average latency curve. A detailed data inspection shows that the latency distribution for Pthreads is heavier-tailed versus the other two systems and somewhat erratic with respect to the number of cores. Consequently, these data points also have a substantially higher variation than others. This behaviour is suspected to be rooted in the Linux kernel scheduler, but a deeper investigation is currently outside of the scope of this work.

The next experiment, reported in Figure 4.11, studies the performance for a varying

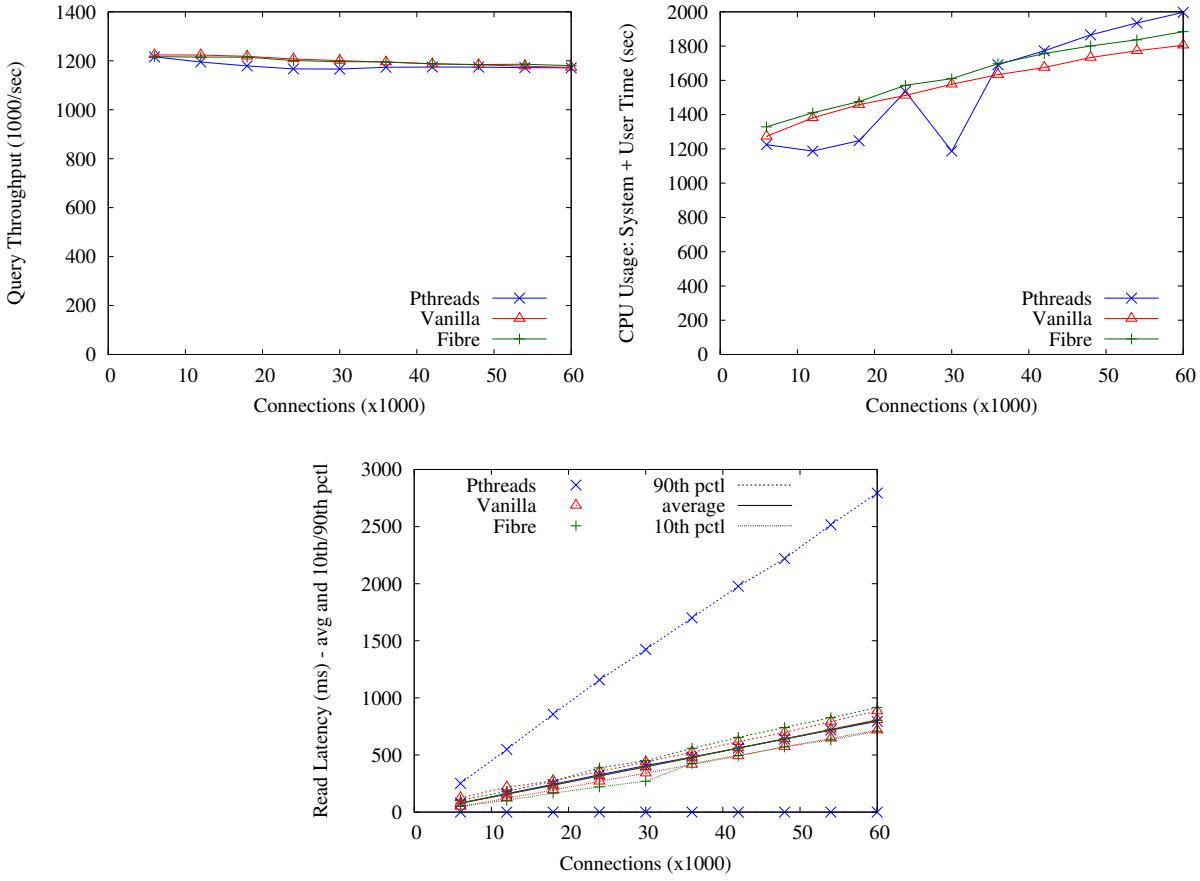


Figure 4.11: Memcached - Connection Scalability (16 cores, write ratio 0.1)

number of connections with a fixed number of 16 cores. Again, all systems have similar throughput, and scale well up to 60,000 connections. Only Pthreads shows erratic results with respect to CPU usage and the data points have a substantially higher variation. Here a bookkeeping problem is suspected in the kernel, although this might also be related to scheduling. Then again, Pthreads' throughput performance in this experiment is indisputable. Its latency again shows a heavier-tailed distribution versus the other two systems, increasingly so with an increasing number of connections. The per-connection memory consumption for the Vanilla implementations is 12 KB, while both threaded versions use about 16 KB memory per connection.

The previous two experiments use a write ratio of 0.1, which is reasonable test value for

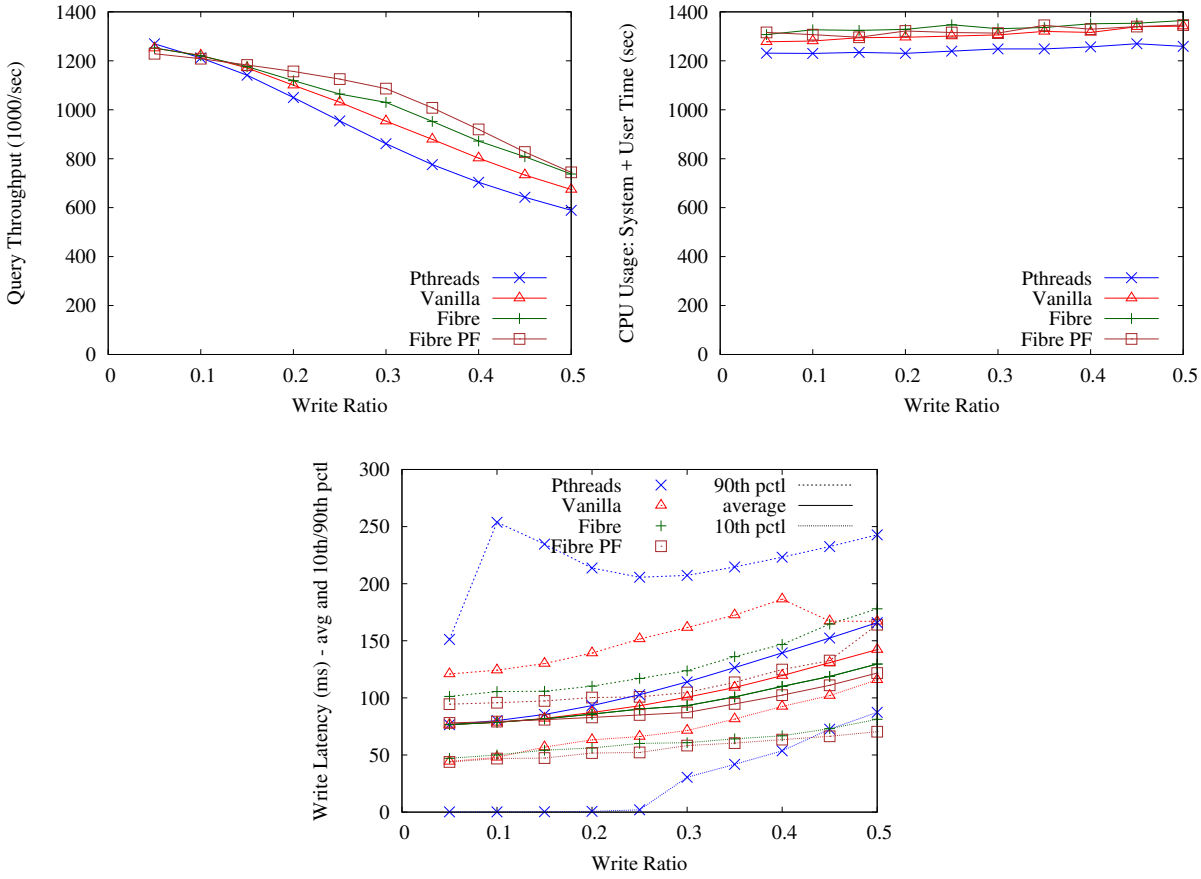


Figure 4.12: Memcached - Read/Write Scalability With Blocking Mutex (16 cores, 6,000 connections)

Memcached, because it is intended as a caching store. However, as a matter of principle, it is also interesting to see how performance changes in relation to this parameter, because it affects the locking intensity in the server. Furthermore, Memcached with an increased write ratio can provide a preliminary estimate of the performance behaviour of a database system with a transactional workload. Thus, in the third experiment the number of cores is fixed at 16 and the number of connections at 6,000, while the write ratio is varied. Moreover, the hierarchical polling design that uses a poller fibre to poll the network is added to the experiments and denoted as 'Fibre PF'. The results are shown in Figure 4.12 and 4.13. Figure 4.12 illustrates the results from the experiments, where all runtimes utilize a blocking mutex for synchronization. Pthreads shows stable results, but CPU usage

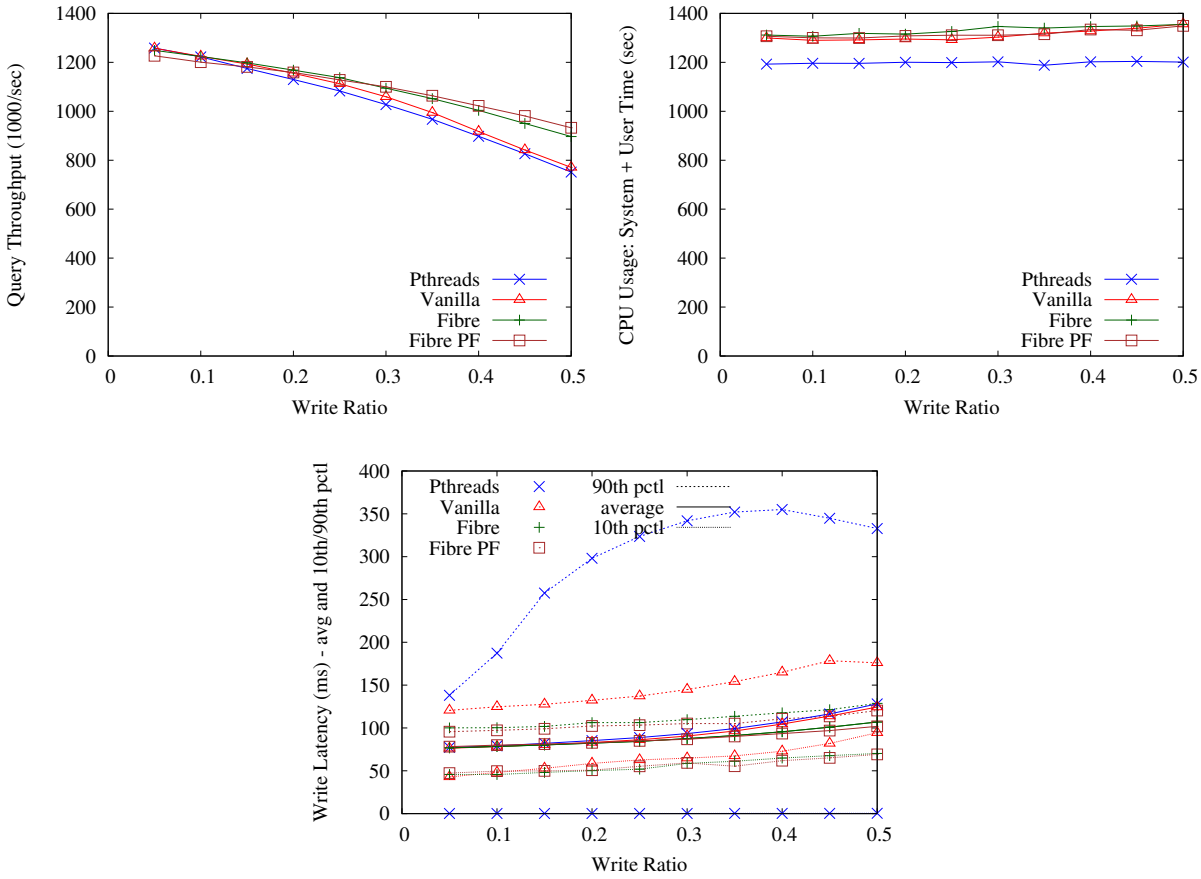


Figure 4.13: Memcached - Read/Write Scalability With Spin-block Mutex (16 cores, 6,000 connections)

increases with a higher write ratio, despite the decline in throughput. Also, a heavier-tailed latency distribution, especially at higher write ratios, is apparent again. The Fibre system with a poller thread performs better than Vanilla at higher write ratios, but at the expense of a heavier-tailed latency distribution. The performance gap is stable and significant. The 'Fibre PF' variation has lower performance at low write ratios but performs better than all other runtimes at higher write ratios with better overall latency. In addition, the 'Fibre PF' variation has slightly better CPU utilization than the version with a poller thread polling in a tight loop.

Figure 4.13 illustrates the results from the same experiment where all runtimes uses a spin-block mutex. A Spin-block mutex spins for a while before blocking the fibre instead of immediately blocking the fibre when the lock is not available. With this version, the Fibre runtime is showing increasingly better performance than the vanilla version at higher write ratio. Further investigation reveals that spinning can avoid blocking in some cases and therefore reduces the number of times the processors run out of work. When processors frequently run out of work, they signal the poller thread more frequently and therefore increase the polling frequency and the polling overhead. In this case, using a spin-block mutex helps to improve the performance. The performance gap between the 'Fibre PF' and the Fibre variations are also smaller than when a blocking mutex is used. CPU usage for both blocking and spin-block cases are very similar to each other. Pthreads still show stable results but it has heavy-tailed latency distribution at high rate ratios.

Fibre Runtime Variations

The last experiment evaluates a few system-level design variations that are possible with the fibre runtime. The first variant uses affinity to pin system threads to cores, while the second variant replaces the default lock-free Nemesis queue data structure with a regular queue protected by a single spin lock. The third variant does not use work-stealing and in the fourth variant, baton passing is added to the user-level blocking mutex locks to pass the ownership among the fibres in a strict FIFO manner. The results, illustrated in Figure 4.14, show that other than the two variants, one without work-stealing and one with baton passing, none of these low-level changes really matter for Memcached's performance, at least for this experiment. The results shows that work-stealing improves the overall performance of the application and lowers the latency by slightly increasing the CPU utilization as the number of cores increases. The baton passing variant has lower throughput, higher latency, and higher CPU usage than other variants as expected. The reason is that most fibres are delayed when the ownership is directly passed to a fibre blocked on the lock. This delay affects the overall performance and latency of the system

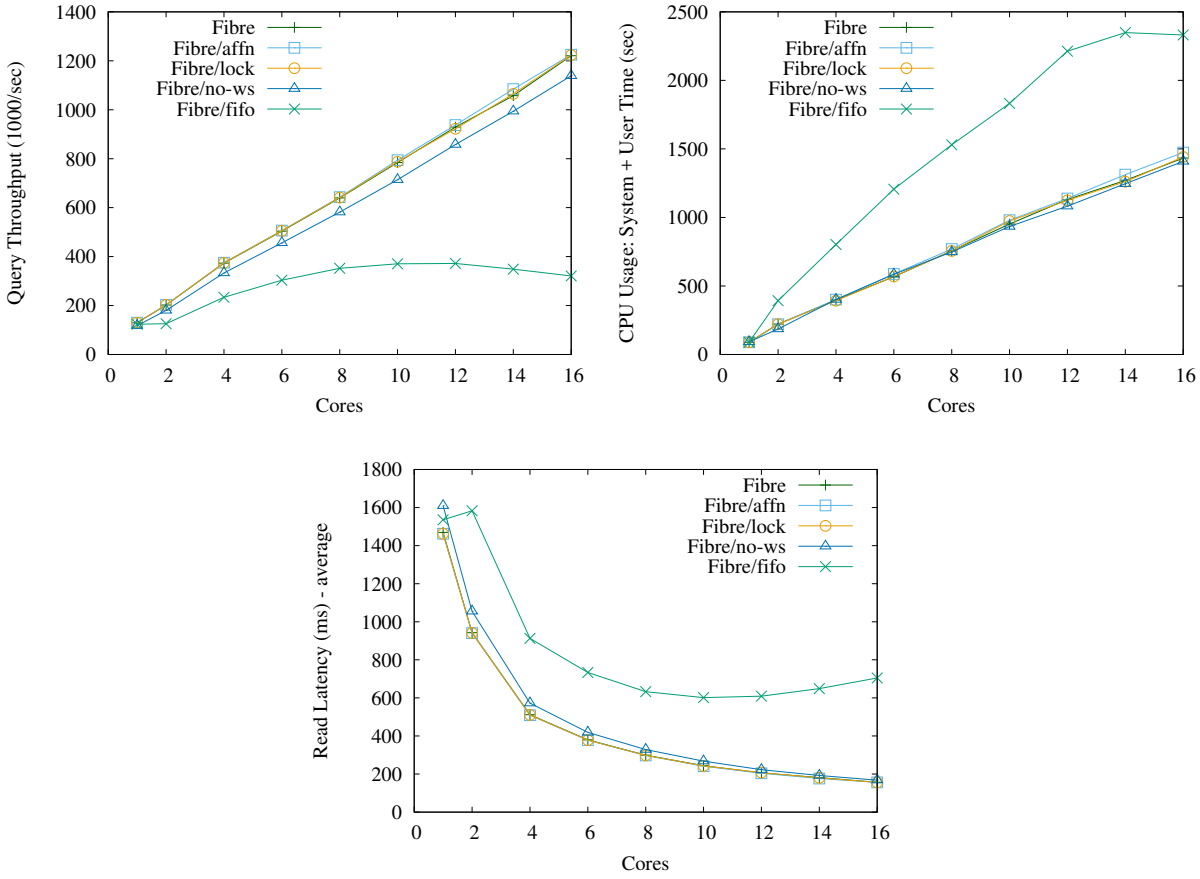


Figure 4.14: Memcached - Runtime Variations (12,000 connections, write ratio 0.1)

as fibres have to wait longer before they can acquire the lock.

Facebook Request Stream

All previous experiments use simple fixed distributions for key size, value size, and inter-arrival time of the requests in the benchmark. However, these values do not reflect real-world applications. Mutilate [119] provides hard-coded distributions for key and value sizes and inter-arrival time of the requests that are extracted from five workloads from Facebook's Memcached deployment [15]. The next experiment uses the same setup as the previous experiments but uses these distributions to evaluate the runtimes using a more

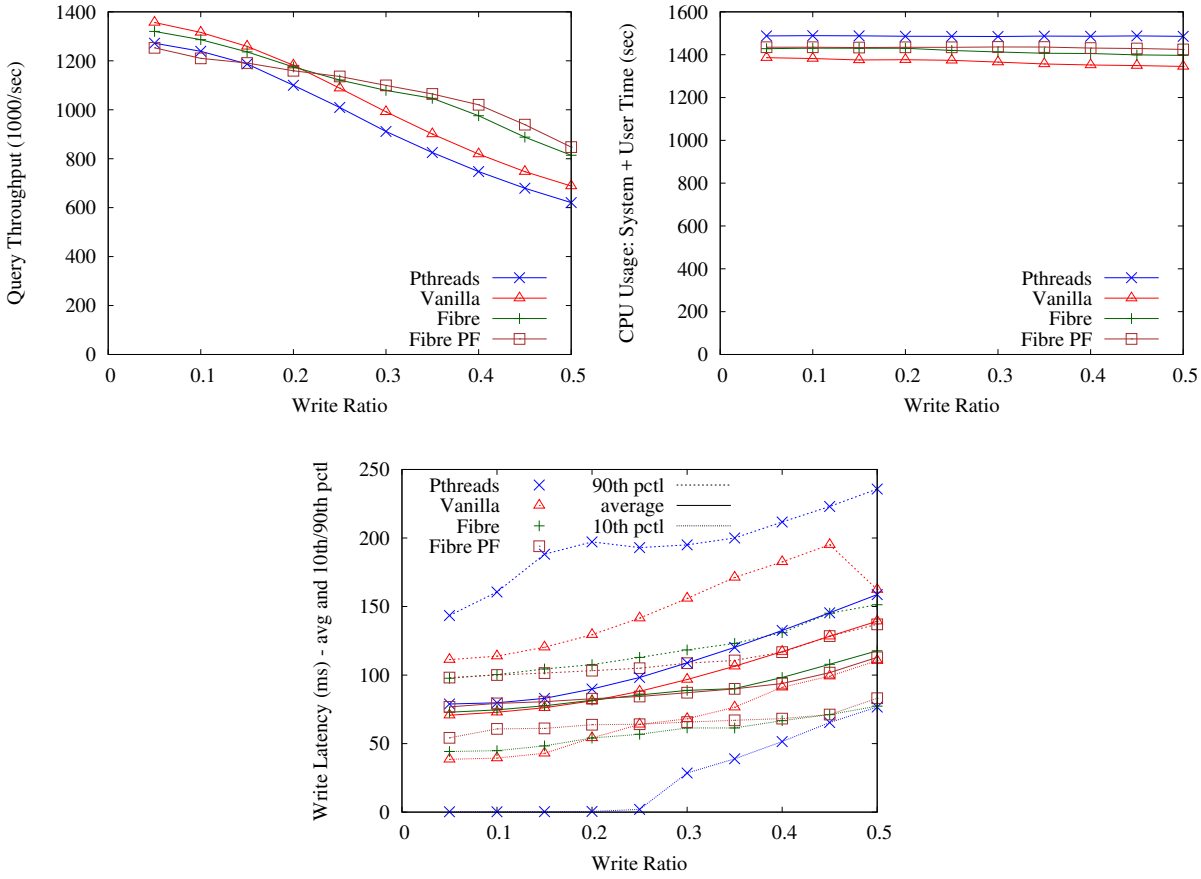


Figure 4.15: Memcached - Facebook Read/Write Scalability With Blocking Mutex (16 cores, 6,000 connections)

realistic workload. The results are shown in Figure 4.15 and Figure 4.16. Overall the latency and throughput results are very similar to the previous experiments. However, Pthreads has higher CPU usage for the case with blocking mutex, and Fibre has lower CPU usage for the case with a spin-block mutex.

The previous experiment is using a pipeline depth of 16 to send 16 GET and SET operations in a single request. However, as discussed in Section 4.3, Fibre does not deliver the same performance as the event-based counterparts when the pipeline depth is set to 1. Therefore, as a matter a principle a new experiment similar to the previous experiment is executed but the pipeline depth is set to one. The results are shown in Figure 4.17

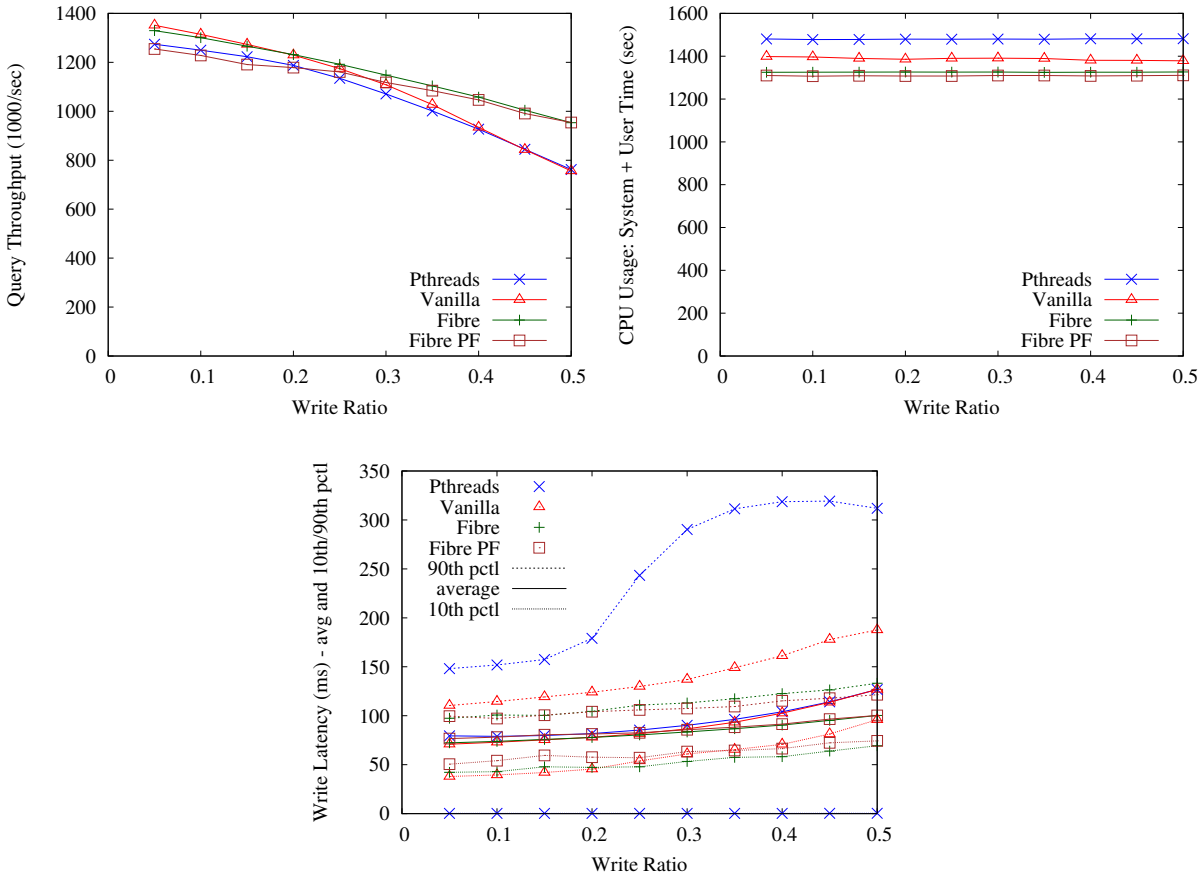


Figure 4.16: Memcached - Facebook Read/Write Scalability With Spin-block Mutex (16 cores, 6,000 connections)

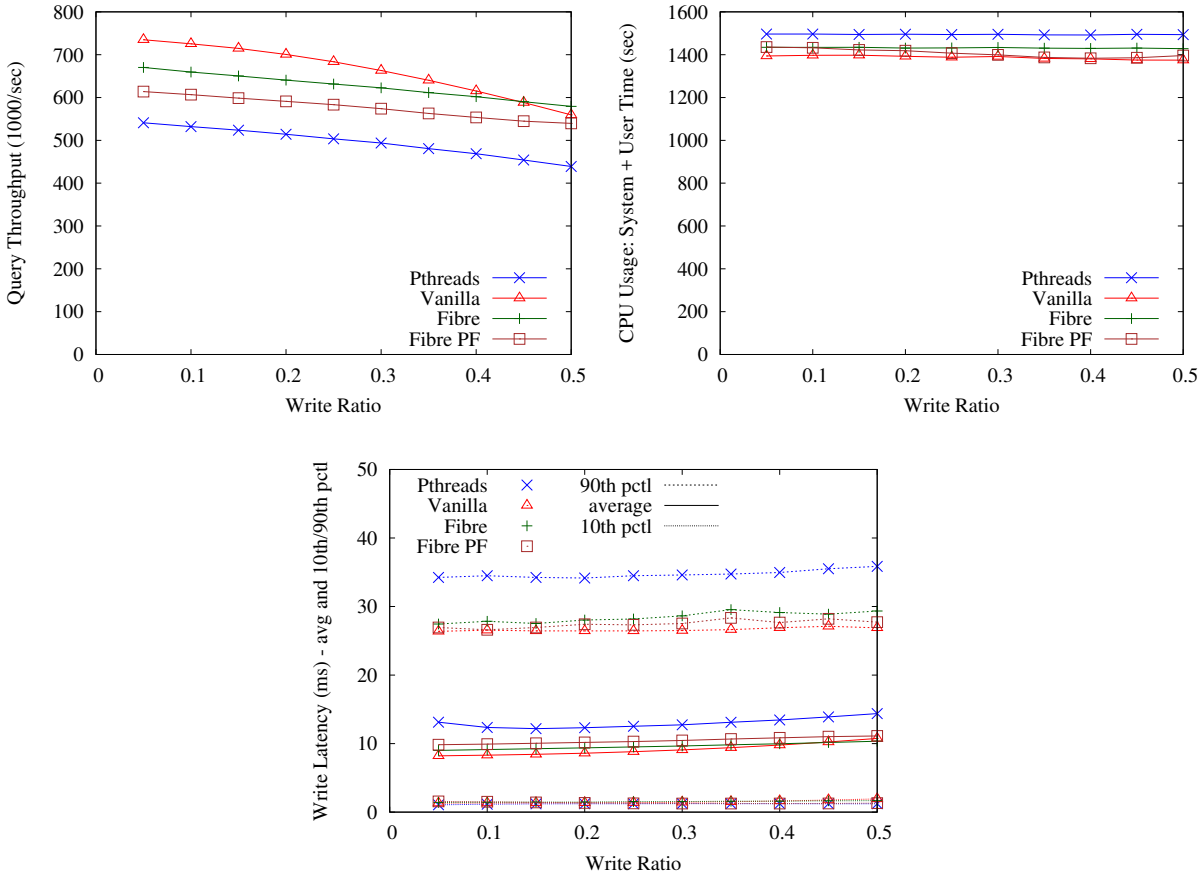


Figure 4.17: Memcached - Facebook Pipeline Depth 1 With Blocking Mutex (16 cores, 6,000 connections)

and 4.18. Indeed, the results show that using the default settings in Fibre, i.e. using a poller thread that polls in a tight loop, the runtime cannot deliver the same throughput as the Vanilla version. In addition, the 'Fibre PF' variant is showing lower throughput and higher average latency in this experiment. Moreover, the 90 percentile latency is also larger than the Vanilla version. Both variants of the Fibre runtime are still doing better than Pthreads both in terms of throughput and latency. Hence, as discussed in Section 4.3, the I/O polling mechanism should be chosen according to the type of workload and the machine that is used to run the application.

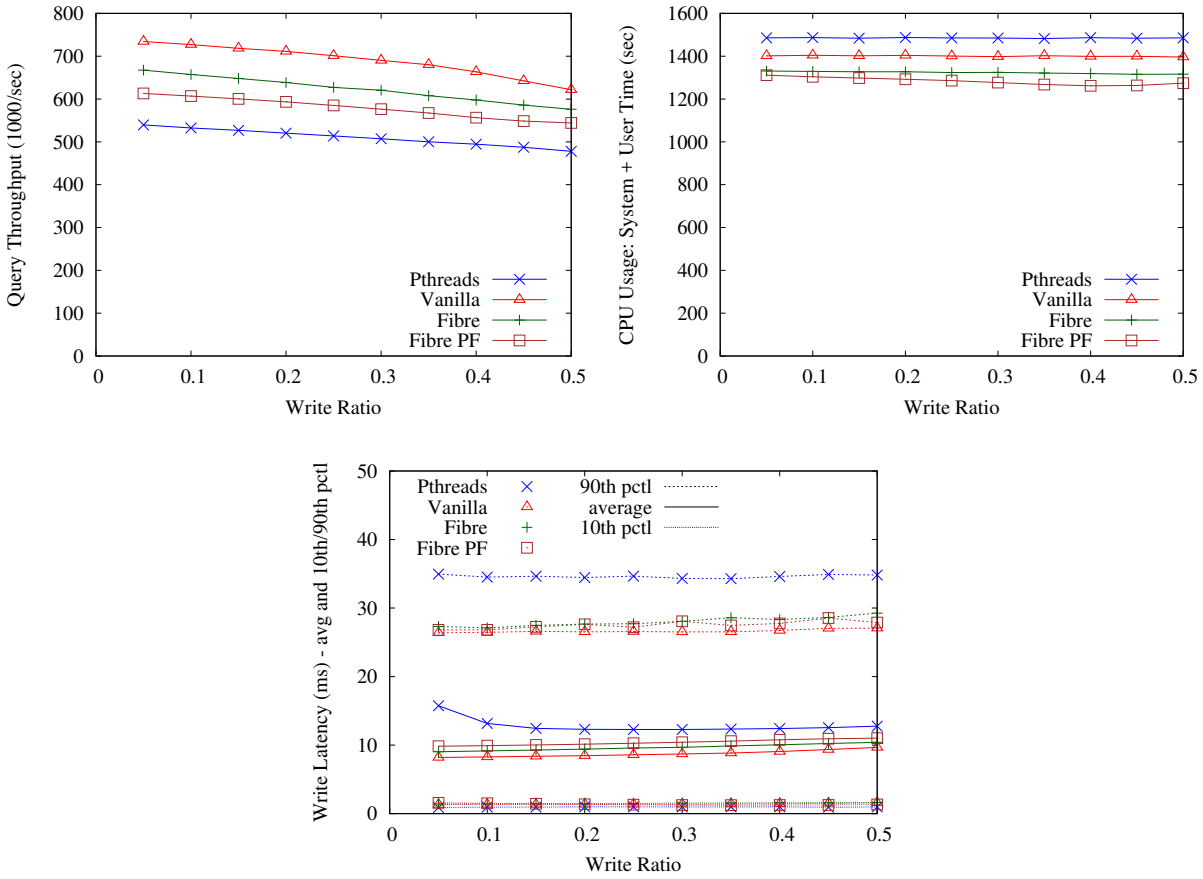


Figure 4.18: Memcached - Facebook Pipeline Depth 1 With Spin-block Mutex (16 cores, 6,000 connections)

Chapter 5

Actor Scheduling

The runtime presented in Chapter 3 re-uses existing concepts, which have been formulated for data-flow programming and/or the actor model, to exploit deferral, batching, and parallelism. The runtime is mainly created to study various design choices for different component of a user-level runtime for threading. However, the results should ultimately be well suited to support programming paradigms such as as actors, CSP, and data-flow programming since the requirements are very similar.

Nonetheless, the behaviour of these systems differ from the behaviour of the network servers that are evaluated using the user-level threading runtime in the previous chapter. For instance, workloads in an actor system are often more dynamic, i.e., actors create other actors, and there are data dependencies among actors, i.e., actors send messages to each other. Consequently, the scheduling of actor objects has different requirements than that of network servers that simply accept connections and serve requests. Hence, this chapter explores the characteristics of actor-based applications that affect the design of a work-stealing scheduler. Furthermore, a locality-aware work-stealing scheduler for actors is proposed and evaluated using benchmarks and a chat server application.

5.1 Related Work

The original work-stealing algorithm uses Randomized Work Stealing (RWS) that indicates when a processor runs out of work, it randomly chooses a victim to steal from. The first randomized work-stealing algorithm for fully-strict computing was introduced in [28]. The algorithm has an expected execution time of $T_1/P + O(T_\infty)$ on P processors, and also

has much lower communication cost than work sharing. For message-driven applications, such as those built with actor-based programming, these bounds can only be regarded as an approximation. The reason is that deep recursion does not occur in event-based actor systems, since computation is driven by asynchronous message passing and cannot be considered as a directed acyclic graph (DAG) [47].

It has been shown that work stealing fundamentally is efficient for message-driven applications [215]. However, random victim selection is not scalable [61], because it does not take into account locality, architectural diversity, and the memory hierarchy [215, 84, 140]. In addition, RWS does not consider data distribution and the cost of inter-node task migration on NUMA platforms [72, 147, 140]. Since there has been very limited work on locality-aware scheduling for actors, this thesis studies and investigates the effect of different locality-aware schedulers on actor frameworks.

There has been very limited research addressing locality-aware or specifically NUMA-aware scheduling for actor runtime systems. Francesquini et al. [72] provide a NUMA-aware runtime environment based on the Erlang virtual machine. They identify *actor lifespan* and *communication cost* as information that the actor runtime can use to improve performance. Actors with a longer lifespan that create and communicate with many short-lived actors are called *hub* actors. The proposed runtime system lowers the communication cost among actors and their hub actor by placing the short-lived actors on the same NUMA node as the *hub* actor, called *home node*. When a worker thread runs out of work, it first tries to steal from workers on the same NUMA node. If unsuccessful, the runtime system tries to migrate previously migrated actors back to that home node. The private heap of each actor is allocated on the home node, so executing on the home node improves locality. As a last resort the runtime steals actors from other NUMA nodes and moves them to the worker's NUMA node.

Although the evaluation results look promising, the caveat, as stated by the authors, is in assuming that hub actors are responsible for the creation of the majority of actors. This hypotheses is a strong assumption and only applies to some applications. Also, when multiple hub actors are responsible for creating actors, the communication pattern among none-hub actors can still be complicated. Another assumption is that all NUMA nodes have the same distance from each other, and the scheduler does not take the CPU cache hierarchy into account. The approach presented takes advantage of knowledge that is available within the Erlang virtual machine, but not necessarily available in an unmanaged language runtime, such as CAF.

In contrast, a locality-aware scheduler is proposed in this chapter that is not based on any assumptions about the communication pattern among actors or information available

through a virtual machine runtime. Instead, it is solely focused on improving performance by improving the scheduler. Also, the full extent and variability of the memory hierarchy is taken into account.

A simple *affinity*-type modification to task scheduling is reported in [216]. Tasks in this system are blocking on a channel waiting for messages to arrive, and thus show similar behaviour to actors waiting on empty mailboxes. In contrast to basic task scheduling, an existing task that is unblocked is never placed on the local queue. Instead, it is always placed at the end of the queue of the worker thread previously executing the task. A task is only migrated to another worker thread by stealing. It is important to note the difference between creating a task and unblocking a task. Newly created tasks are still placed on the local queue, but unblocking existing tasks places them on the local queue of the processor they were blocked on.

This modification leads to significant performance improvements for certain workloads and thus contradicts assumptions about treating the queues in LIFO manner. However, the system has a well-defined communication pattern and long task lifespans, in contrast to an actor system that is non-deterministic with a mixture of short- and long-lived actors. Using a hierarchical scheduler both LIFO and affinity policies are implemented and evaluated, and the results are presented in Section 5.4.

Wolke et al. [222] present modifications to the CAF scheduler using LIFO and affinity policies. The details of the implementation slightly differs from what is presented here but ultimately the authors present similar results to this work and draw similar conclusions. However, the authors mainly compare CAF against other frameworks and different policies are only evaluated by benchmarks. In contrast, the focus of this work is to compare these policies against each other and evaluate them using both benchmarks and a large-scale real world application. It is shown later in this chapter that the conclusions drawn from the benchmarks does not apply to a large-scale application due to added contention.

Various locality-aware work-stealing schedulers have been proposed for other parallel programming models and shown to improve performance. Suksompong et al. [188] investigate localized work-stealing and provide running time bounds when workers try to steal their own work back. Acar et al. [5] study the data locality of work-stealing scheduling on shared-memory machines and provide lower and upper bounds for the number of cache misses, and also provide a locality-guided work-stealing scheduling scheme. Chen et al. [48] present a cache-aware two-tier scheduler that uses an automatic partitioning method to divide an execution DAG into the inter-chip tier and the intra-chip tier.

Olivier et al. [146] provide a hierarchical scheduling strategy where threads on the same chip share a FIFO task queue. In this proposal, load balancing is performed by work

sharing within a chip, while work-stealing only happens among chips. In follow-up work, the proposal is improved by using a shared LIFO queue to exploit cache locality between sibling tasks as well as between a parent and newly created task [147]. Moreover, the work-stealing strategy is changed, so that only a single thread can steal work on behalf of other threads on the same chip to limit the number of costly remote steals. Pilla et al. [156] propose a hierarchical load-balancing approach to improve the performance of applications on parallel multi-core systems and show that Charm++ can benefit from such a NUMA-aware load balancing strategy.

Min et al. [140] propose a hierarchical work-stealing scheduler that uses the Hierarchical Victim Selection (HVS) policy to determine from which thread a thief steals work, and the Hierarchical Chunk Selection (HCS) policy that determines how much work a thief steals from the victim. The HVS policy relies on the scheduler having information about the memory hierarchy: cache, socket, and node (this work also considers many-core clusters). Threads first try to steal from the nearest neighbours and only upon failure move up the locality hierarchy. The number of times that each thread tries to steal from different levels of the hierarchy is configurable. The victim selection strategy presented here in Section 5.3 is similar to HVS, but takes NUMA distances into account. The HCS policy allows stealing a chunk of tasks from the task queue of the victim thread on another node for better scalability. However, within the same node, small chunk sizes (1 or 2) have been shown to provide sufficiently good performance.

Drebes et al. [63, 64] combine topology-aware work-stealing with work pushing and dependence-aware memory allocation to improve NUMA locality and performance for data-flow task parallelism. Work pushing transfers a task to a worker whose node contains the task's input data according to some dependence heuristics. Each worker has a MPSC FIFO queue in addition to a work-stealing deque. The MPSC queue is only processed when the deque is empty. However, this approach is not applicable to a latency-sensitive actor application for two reasons: first, the actor model is non-deterministic and data dependence difficult to infer at runtime. Second, adding a lower-priority MPSC queue adds complexity and can cause some actors to be inactive for a long time, which violates fairness and thus causes long tail latencies for the application.

Moreover, the proposed deferred memory allocation relies on knowing the task dependencies in advance, which is not possible with the actor model. Also, producers write data to data buffers of each task and the proposed algorithm defers the allocation of each data buffer to when the node of the worker that executes the producer is known. This approach avoids costly remote write accesses and allows producers to write locally, which results in significant performance improvements. However, senders of messages in the actor model create the messages locally and only place a pointer in the single mailbox of the receiver.

Therefore, this optimization cannot be applied to the actor model. The topology-aware work-stealing introduced by this work is similar to the work presented here, but it is evaluated in combination with deferred memory allocation and work-pushing. Thus, it is not possible to discern the isolated contribution of topology-aware work-stealing.

Majo et al. [130] point out that optimizing for data locality can counteract the benefits of cache contention avoidance and vice versa. In Section 5.4 results that demonstrate this effect for actor workloads are presented where aggressive optimization for locality increases the last-level cache contention.

5.2 Characteristics of Actor Applications

Key operations can be slowed down when an actor migrates to another core on a NUMA system, depending on the NUMA distance. This performance degradation can come from messages that arrive from another core, or from accessing the actor's state that is allocated on a different NUMA node. Depending on the type of actor and the communication pattern, the amount of degradation differs. Therefore, improving locality does not benefit all workloads. The following factors in applications and workloads for actor-based programming are identified that can affect the performance of a work-stealing scheduler on a hierarchical NUMA machine:

1. *Memory allocated for actor and access pattern*: Actors sometimes only perform computations on data passed to them through messages. For simplicity, actors that only depend on message data are denoted as *stateless* actors, and actors that do manage local state are denoted as *stateful* actors.

   Stateful actors allocate local memory upon creation and access it or perform other computations depending on the type of a message and their state when they receive a message. A stateful actor with sizable state and intensive memory access to that state is better executed closer to the NUMA node where it is created. Also, for better cache locality, especially if the actor receives messages from its spawner frequently, it is better to keep such an actor on the same core, or a core that shares a higher-level CPU cache with the spawner core. The reason is that those messages are hot in the cache of the spawner actor.

   On the other hand, stateless actors do not allocate any local memory and can be spawned multiple times for better scalability. For such actors, the required memory to process messages is allocated when they start processing a message and deallocated

when the processing is done. Therefore, the only substantial memory that is accessed is the one allocated for the received message by the sender of that message. Such actors are better executed closer to the message sender.

2. *Message size and access pattern*: The size of messages has a direct impact on the performance and locality of actors on NUMA machines. Messages are allocated on the NUMA node of the sender, but accessed on the core that is executing the receiver actor. If the size of messages is typically larger than the size of the local state of an actor, and the receiving actor accesses the message intensively, actors are better to be activated on the same node as the sender of the message.

3. *Communication pattern*: Since the actor model is non-deterministic, it is difficult to generally analyze the communication pattern among actors. Two actors that are sending messages to each other can go through different states and thus have various memory-access patterns. In addition, the type and size of each message can vary depending on the state of the actor. No assumptions is made about the communication pattern of actors, unlike others [72].

Aside from illustrating the trade-offs involved in actor scheduling, these observations are also useful to determine which benchmarks realistically demonstrate the benefits of locality-aware work stealing schedulers and which represent worst-case tests.

5.3 Locality-Aware Scheduler (LAS)

The locality-aware scheduler consists of three stages: memory hierarchy detection, work placement, and work stealing. When an application starts running, the scheduler determines the memory hierarchy of the machine. Also, a new actor is placed on the local or a remote NUMA node depending on the type of the actor. Finally, when a worker thread runs out of work, it uses a locality-aware policy to steal work from another worker thread.

5.3.1 Memory Hierarchy Detection

The work-stealing algorithm needs to be aware of the memory hierarchy of the underlying system. In addition to the cache and NUMA hierarchy, differing distances between NUMA nodes are an important factor in deciding where to steal tasks from. Access latencies can vary significantly based on the topological distance between access node and storage node.

The scheduler builds a representation of the locality hierarchy using the *hwloc* library [36] that uses hardware information to determine the memory hierarchy, which the scheduler represents as a tree. The root is a place-holder representing the whole system, while intermediate levels represent NUMA nodes and groups, taking into account NUMA distances. Subsequent nodes represent shared caches and the leaves represent the cores on the system. This representation is independent from any particular hardware architecture.

Actor Placement

For fully-strict computations, data dependencies of a task only go to its parent. Thus the natural placement for new tasks is the core of the parent task. However, actors can communicate arbitrarily and thus, local placement of newly created actors does not guarantee the best performance. For example, actors receiving remote messages pollute the CPU cache for actors that execute later and process messages from their parents. Also, as stated earlier, depending on the size of the message in comparison to state variables, placing the actor in the sender's NUMA node can help or hurt performance. Determining the best strategy at runtime without future information can add significant overhead, and there is no apparent optimal approach.

The exception are *hub actors* [72], i.e., long-living actors that spawn many children and communicate with them frequently. Such actors place high demand on the memory allocator and can interfere with each other if placed on the same NUMA node. Furthermore, if a locality-aware affinity policy tries to keep actors on their home node, placing multiple hub actors on the same NUMA node further increases contention over shared resources and thus reduces the performance. Hence, the scheduler uses the same algorithm for initial placement of *hub actors* [72] to spread them across different NUMA nodes. The programmer needs to annotate hub actors. The system then tags corresponding structures at compile time and the runtime scheduler uses this information to place such actors far from each other.

5.3.2 Locality-aware Work-stealing

A locality-aware victim selection policy attempts to keep tasks closer to the core that created them or was running them previously to take advantage of better cache locality. Depending on the hardware architecture, cores might share a higher-level CPU cache. Therefore, in the scheduler, the thief worker thread first steals from worker threads executing on nearby cores in the same NUMA node with shared caches to improve locality. If

there is no work available in the local NUMA node, the hierarchical victim selection policy tries to steal jobs from worker threads of other NUMA nodes with increasing NUMA distance. The goal of NUMA-aware work stealing is to avoid migrating actors among NUMA nodes to the extent possible, and thus to remove the need for remote memory accesses.

Limiting the worker threads to initially choose their victims within their own NUMA node can lead to more frequent contention over deques on the local NUMA node in comparison to using the random victim selection strategy. For example, if a single queue still has work, while all other worker threads run out of work, is the worst-case scenario for a work-stealing scheduler. However, this case appears frequently in actor applications, where a hub actor creates multiple actors and other worker threads steal from the local deque of the thread that runs the hub actor. Further investigations shows that when stealing fine-grained tasks with workloads that are 20μs or shorter, the performance penalty ratio increases exponentially as the number of thief threads increase. For more coarse-grained tasks, the performance penalty is not significant, since the probability of contention decreases.

To alleviate this problem, the scheduler keeps track of the number of threads per NUMA node that are polling the local node. This number is used along with the approximate size of the deques in the node to reduce the number of threads that are simultaneously polling a deque (see Algorithm 2). If there is only a single non-empty deque and more than half of threads under that node are polling that deque, the thief thread backs off and tries again later.

In addition, polling the queue of many other worker threads with empty queues can result in wasting CPU cycles when the number of potential victims is limited. In CAF, a worker thread constantly polls its own deque and after a certain number of attempts, polls a victim deque. To avoid wasting CPU cycles, the deque is modified and an approximate size of the deque using a counter is added to it. A thief uses this approximate size when it attempts to steal from other workers executing on the same NUMA node. If there are non-empty queues, it chooses one randomly, otherwise if all the queues are empty, the thief immediately moves up to the next higher level (Algorithm 2). This approach removes the overhead of polling empty queues on the local NUMA node and thus decreases the number of wasted CPU cycles. Since there is a fixed number of cores on a NUMA node, scanning their queue sizes adds little overhead that remains constant even when the application scales.

When a worker runs out of work, it becomes a thief and uses the memory hierarchy tree provided by the scheduler to perform hierarchical victim selection as described in Algorithm 2. The updated vertex v is passed to the function each time to complete the

tree traversal. An empty result or a victim with an empty deque means that the thief has to try again.

Algorithm 2 Hierarchical Victim Selection

```
1:  T: Memory hierarchy tree
2:  C: Set of cores under v
3:  p: Number of threads polling under local NUMA node
4:  r: Number of steal attempts for v
5:  procedure ChooseVictim(v)
6:      if r = Size(C) and v ≠ root(T) then
7:          v ← parent(v)
8:      if v is in local NUMA node then
9:          S = {s | all non-empty local deques}
10:         if S = ∅ then
11:             v ← parent(v)
12:         else if size(S) = 1 and p > size(C)/2 then
13:             return ∅
14:         else return random from S
15:     return random from C
```

Two variants of the Locality-Aware Scheduler (LAS) are created that differ in their placement strategy. When an existing actor is unblocked by a message, the *local* variant (Locality-Aware Scheduler/Local (LAS/L)) places the actor on the local deque, while the *affinity* variant (Locality-Aware Scheduler/Affinity (LAS/A)) places the actor at the end of the deque of the worker thread previously executing it. In both cases, newly created actors are pushed to the head of the local deque. LAS/L is similar to typical work-stealing placement where all activated and newly created tasks are pushed to the head of the local deque. LAS/A improves actor-to-thread (and thus to-core) affinity, because actors are moved only by stealing. However, it adds overhead to saturated workers and increases contention when placing actors on remote deques, which is further discussed in Section 5.4.3.

5.4 Experiments and Evaluation

Experiments are conducted on an Intel and an AMD machine with different NUMA and memory hierarchies described in Chapter 4. The experiments are performed with CAF version 0.12.2, compiled with GCC 5.4.0, on Ubuntu Linux 16.04 with kernel 4.4.0.

The experiments compare CAF's default Randomized Work Stealing (RWS) scheduler with LAS/L and LAS/A. First, the performance is evaluated using benchmarks to study the effect of scheduling policy on different communication patterns and message sizes. Next, a simple chat server is used to observe the efficiency of schedulers for an application that has a large number of actors with non-trivial communication patterns, different behaviours, and various message sizes.

5.4.1 Benchmarks

The first set of experiments attempts to isolate the effects of each scheduling policy for different actor communication patterns. A subset of benchmarks from the BenchErl [13] and Savina [98] benchmark suites is chosen that represents typical communication patterns used in actor-based applications. Some of these benchmarks are adopted from task-parallelism benchmarks, but modified to fit the actor model.

- *Big (BIG):* In a many-to-many message passing scenario many actors are spawned and each one sends a ping message to all other actors. An actor responds with a pong message to any ping message it receives.
- *Bang (BANG):* In a many-to-one scenario, multiple senders flood the one receiver with messages. Senders send messages in a loop without waiting for any response.
- *Logistic Map Series (LOGM):* A synchronous request-response benchmark pairs control actors with compute actors to calculate logistic map polynomials through a sequence of requests and responses between each pair.
- *All-Pairs Shortest Path (APSP):* This benchmark is a weighted graph exploration application that uses the Floyd-Warshall algorithm to compute the shortest path among all pairs of nodes. The weight matrix is divided into blocks. Each actor performs calculations on a particular block and communicates with the actors holding adjacent blocks.
- *Concurrent Dictionary (CDICT):* This benchmark maintains a key-value store by spawning a dictionary actor with a constant-time data structure (hash table). It also spawns multiple sender actors that send write and read requests to the dictionary actor. Each request is served with a constant-time operation on the hash table.
- *Concurrent Sorted Linked-List (CSLL):* This benchmark is similar to CDICT but the data-structure has linear access time (linked list). The time to serve each request

depends on the type of the operation and the location of the requested item. Also, actors can inquire about the size of the list, which requires iterating through all items.

- *NQueens first N Solutions (NQN)*: A divide-and-conquer style algorithm searches a solution for the problem: "How can N queens be placed on an $N \times N$ chessboard, so that no pair attacks each other?"

- *Trapezoid approximation (TRAPR)*: This benchmark consists of a master actor that partitions an integral and assigns each part to a worker. After receiving all responses they are added up to approximate the total integral. The message size and computation time is the same for all workers.

- *Publish-Subscribe (PUBSUB)*: Publish/subscribe is an important communication pattern in actor programs that is used extensively in many applications, such as chat servers and message brokers. This benchmark is implemented using CAF's group communication feature and measures the end-to-end latency of individual messages. It represents a one-to-many communication pattern where a publisher actor sends messages to multiple subscribers. Actors can subscribe to more than one publisher.

5.4.2 Experiments

The benchmark results are shown in Figure 5.1. The execution time is the average of 10 runs and normalized to the slowest scheduler (lower is better). All experiments are configured to keep all cores busy most of the time, i.e., the system operates at peak load.

The RWS scheduler performs relatively better for the BIG benchmark and it outperforms both LAS/L and LAS/A. This benchmark represents a symmetric many-to-many communication pattern where all actors are sending messages to each other. This workload benefits from a symmetric distribution of work. Other experiments (not shown here) show that using the NUMA *interleave* memory allocation policy improves the performance further. For this particular workload, improving locality does not translate to improving the performance.

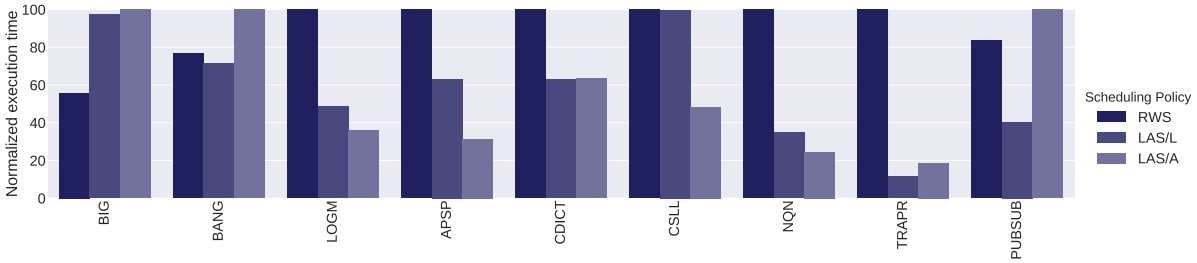
The BANG benchmark represents workloads using many-to-one communication. Messages have very small sizes and no computation is performed. Since the receiver's mailbox is the bottleneck, improving locality does not significantly affect the overall performance. LAS/L only improves the performance slightly by allocating more messages on the local node.

LOGM and APSP both create multiple actors during startup and each actor frequently communicates with a limited number of other actors. In addition, computation depends on an actor's local state and message content. For both workloads, LAS/L and LAS/A outperform RWS by a great margin. In such workloads, each actor can only be activated by one of the actors it communicates with. If one of the communicating actors is stolen and executes on another core, in RWS and LAS/L, it causes the other actors to follow and execute on the new core upon activation. Since all actors maintain local state that is allocated upon creation of the actor, all actors that are part of the communication group experience longer memory access times if one of them migrates to another NUMA node. Keeping actors on the same NUMA node and closer to the core they were running before can prevent this.

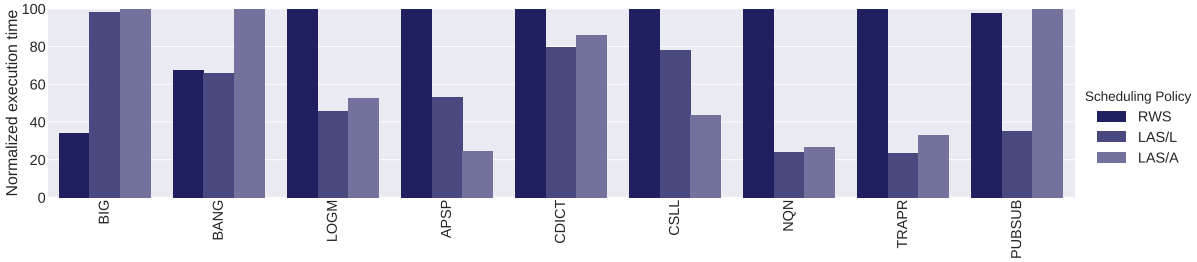
LAS/A improves performance further by preventing other actors from migrating along with the stolen actor. Even though actors are occasionally moved to another NUMA node, the rest of the group stays on their own NUMA node. Thus, LAS/A performs better than LAS/L by preventing group migration of actors. For LOGM, since each pair of actors is isolated from other actors, stealing one actor translates to moving one other actor along with it. However, in APSP each actor communicates with multiple actors, which means stealing one actor can cause a chain reaction and several actors that do not directly communicate with the stolen actor might also migrate. LAS/A therefore has a stronger effect on APSP than LOGM.

CDICT and CSLL represent workloads where a central actor spawns multiple worker actors and communicates with them frequently. The central actor is responsible for managing a data structure and receives read and write requests from the worker actors. CDICT benefits from improved locality provided by both locality-aware schedulers. Since the majority of operations are allocating and accessing messages between the central actor and the worker actors, placing worker actors closer to the central actor leads to improved performance due to faster memory accesses. In CDICT all requests are served from a hash table in roughly constant time. In such a setting, LAS/A can cause an imbalance in service times, because some actors are being placed on cores with higher memory access times. Since the service time for each request is fairly small, this additional overhead can slightly slow down the application.

However, in CSLL, LAS/A outperforms LAS/L and RWS. First, the overhead that LAS/A imposes on the central actor becomes negligible in comparison with the linear lookup time into the linked list. Because of the resulting increased service times, most actors ultimately become inactive, waiting for a response from the central actor. The corresponding worker threads end up being idle and seeking work. With LAS/A, the response unblocks a worker actor on its previous worker thread, so that execution can



(a) AMD



(b) Intel

Figure 5.1: Scheduler Comparison

continue right away. However, LAS/L unblocks worker actors on the same worker thread as the central actor. This introduces additional latency until the worker actor executes or alternatively, until it is stolen by an idle worker thread.

NQN is a divide-and-conquer algorithm where a master actor is responsible for dividing the work among a fixed number of worker actors. Each worker actor performs a recursive operation on the task assigned to it and further divides the task to smaller subtasks. But instead of spawning new actors, it reports back to the master actor that assigns the new tasks to the worker actors in a round-robin fashion. Therefore, all worker actors are constantly producing and consuming messages. The computation performed for each message depends on the content of the message and all items in each message are accessed during computation. Improving locality and placing worker actors closer to each other and to the master actor has a significant impact on performance. LAS/L and LAS/A perform 5 times faster than RWS in this case.

In TRAPAR, worker actors receive a message from a master actor, perform some calculations, send back a message, and exit. LAS/L and LAS/A improve the performance up to 10 times for this benchmark. Since all actors are created on the local deque of the master actor, and the tasks are very fine-grained, locality-aware scheduling increases the

chance of local cores to steal and run these tasks closer to the master actor. Since all communications are with the master actor, the performance is improved significantly.

The PUBSUB benchmark shows significant end-to-end message latency improvement when the LAS/L policy is used in comparison with RWS. LAS/L keeps the subscribers closer to the publisher that sends them a message and improves the locality. However, LAS/A shows worse performance than the other two policies. Profiling the code reveals that worker threads are stalled by lock contention most of the time. The reason is that with LAS/A, worker threads place the newly created tasks on the deque of other cores rather than the local core. Since publishers are constantly unblocking actors on other cores, this leads to higher contention when there are large number of publishers and subscribers.

There are minor differences between the results from the AMD machine and the Intel machine. These differences come from the differences in the NUMA setup of each machine, explained at the beginning of this section. The probability that RWS moves tasks to a NUMA node with higher access times is higher for the AMD machine. Thus, locality-aware schedulers are slightly more effective on the AMD machine.

In general, the results indicate that workloads with many-to-many communication patterns (BIG) do not benefit from locality-aware schedulers. Workloads where actors are communicating with a small cluster of other actors (LOGM and APSP), actors communicate with a central actor and access message contents (CDICT and CSLL), or actors communicate with a central actor one or multiple times and perform computations that depend on the content of the message (NQN and TRAPR), benefit from locality-aware schedulers. Moreover, in most cases where locality improves performance, LAS/A performs similar or better in comparison with LAS/L. However, in one case (PUBSUB) LAS/A causes high contention and a performance decrease.

Another experiment is performed to study the effect of message size on the performance of locality-aware schedulers. The CDICT benchmark is modified to make the *value* size configurable for each key-value pair. This affects the size of messages and the size of memory operations performed by the central actor. Worker actors submit write requests 20% of the time. Figure 5.2 shows the results for this experiment executed on the AMD machine. Experiments on the Intel machine show similar results, so are not shown here.

The results show that the LAS/L scheduler outperforms both RWS and LAS/A for value sizes smaller than 256 words. LAS/L improves locality and since most messages and objects fit into lower level caches (L1 and L2), improved locality further improves the performance. RWS distributes tasks among NUMA nodes and therefore imposes higher memory access times. LAS/A also adds additional overhead, because it causes the dictionary actor to unblock some actors on remote NUMA nodes. However, as the value size gets larger,

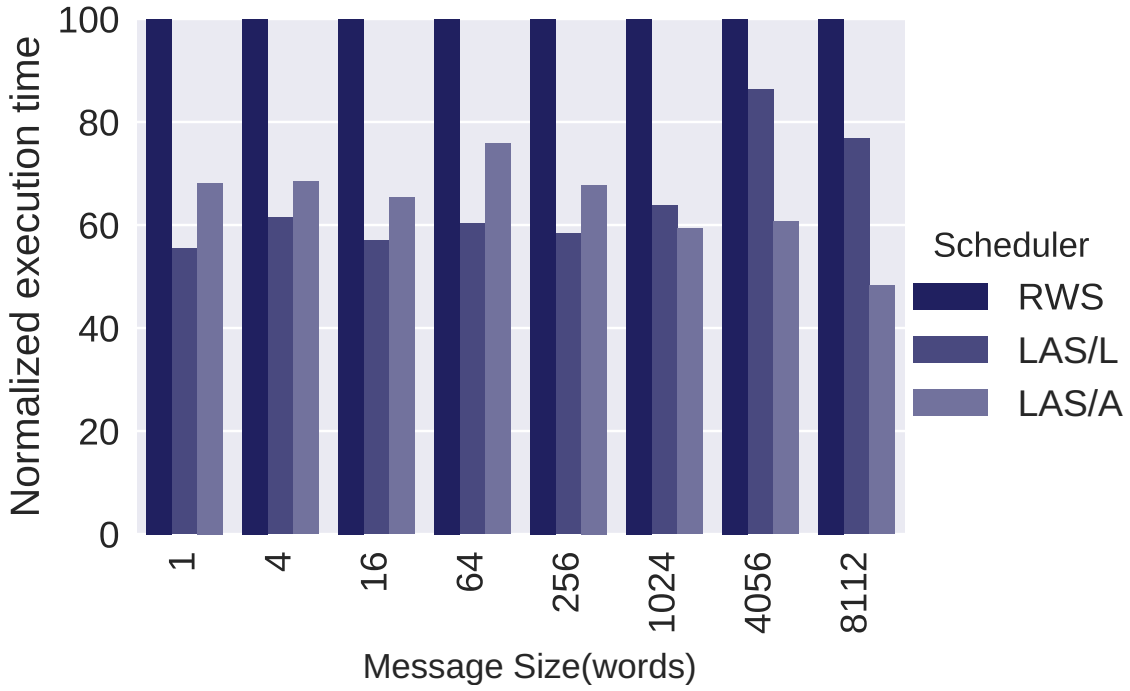


Figure 5.2: Effect of Message Size on Performance

messages and objects do not fit into lower level caches. Since LAS/L keeps most actors on the same NUMA node as the dictionary actor, this creates contention in the L3 cache, which slows down the dictionary actor. LAS/A, on the other hand, distributes actors to other NUMA nodes as well, which avoids the contention in L3, such that remote access overhead is compensated by lower contention. RWS also avoids the contention problem and therefore the difference between LAS/L and RWS decreases. In fact, when increasing the percentage of write requests, RWS can even outperform LAS/L as measured by another experiment not presented here.

5.4.3 Chat Server

To evaluate both variants of LAS using a more realistic scenario, a chat server similar to [164] is implemented that supports one-to-one and group chats. Each user (session) is represented by an actor that holds the state for the session in the server application. Chat

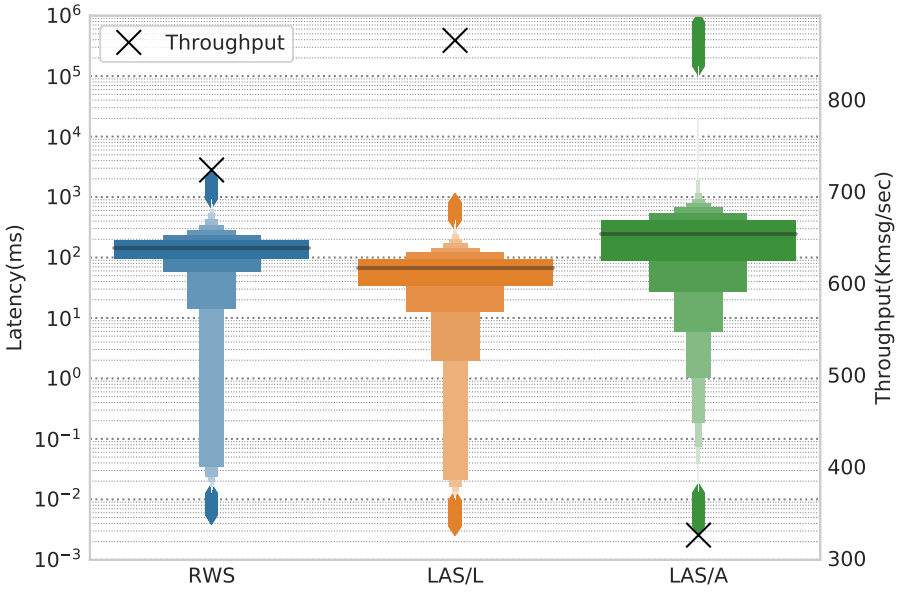
groups are created using the publish/subscribe based group communication in CAF. To simplify the implementation, the server does not include network operations and the workload is generated and consumed in the same process. However, the chat server implements pre- and post-processing operations that would normally be carried out in the context of communication with remote clients, such as encryption.

Each user has a friend list, group list, and blocked list, which represent the corresponding lists of users respectively. Information about each session and a log of messages is stored in an in-memory key-value storage controlled by a database actor. In addition, each session actor stores its information in a local cache controlled by a local cache actor. When a session actor receives a message, it first decrypts the message, uses the receiver user ID to find the reference to the receiving actor, and forwards the message to that actor. The message is also logged in both the local cache and the central storage. When the receiver actor receives the message, it first checks whether the sender is in its blocked list. If not, it encrypts the message as if it was sent out to a remote client. If a message is sent to a group, an actor representing the group forwards the message to all subscribers, which creates a one-to-many communication pattern.

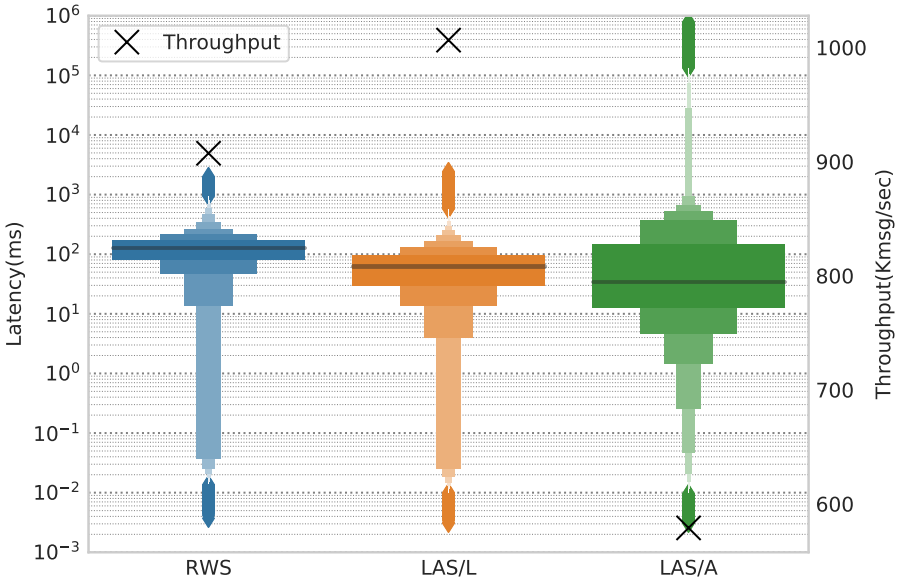
The chat server is configured to run with 1 million actors and 10000 groups. Each user has a random number of friends (max. 100), subscribes to a random number of groups (max. 10), and blocks a random number of users (max. 5). Random numbers are uniformly distributed, and all session and group actors are spawned before the experiment starts. To simulate receiving messages from users, every second each session actor uses a timer to send a message with a random length (max 1024 bytes) to a randomly chosen user or group from its friend or group list. 5% of the messages are sent to groups and the rest are direct messages. The experiment drives each system to peak load. The overall throughput is measured along with the end-to-end latency of each message from the moment the sender actor (on the server side) generates the message until the moment the receiver actor is ready to send it to the remote client.

Figure 5.3 uses letter-value plots to show the distribution of latencies using different scheduling policies on the AMD and Intel machines. LAS/L has higher throughput and better latency distribution than the other two policies on both AMD and Intel machines. LAS/A has significant lower throughput than both RWS and LAS/L and the latency distribution shows a higher average latency with a long latency tail that goes up to 1000 seconds on both machines. Profiling indicates that the workload is dominated by deque lock contention, similar to PUBSUB. This contention causes variable and high latency numbers, and fairly low throughput, because threads are relatively often blocked on a lock.

Hence, although LAS/A is performing fairly well in simple scenarios and benchmarks,



(a) AMD Opteron



(b) Intel Xeon

Figure 5.3: Throughput and End-to-end Message Latency Distribution of the Chat Server.

lock contention significantly affects its performance when the application scales and number of actors increases. Therefore, the proposal in [216] does not apply to large-scale actor-based applications. The LAS/L policy, on the other hand, shows stable performance improvements over RWS and reduces the latency.

Chapter 6

Utilization Model

This chapter aims to use a simplified model to determine the effects of lock-based mutual exclusion on concurrent software. Particularly, the effect of lock contention on utilization and the limits of increasing concurrency in applications with critical sections.

The effects of lock contention on scalability are known in principle and have been studied before. Pan et al. [152] exploit queueing networks to model lock contention in applications with locks. They use Mean Value Analysis (MVA) to predict lock contention; this model is explained further later in this section. Dundas et al. [66] propose a probability model to analyze the performance of multi-threaded locking in bucket hash tables. The authors use a similar queueing model to Pan et al. [152] but directly calculate the contention probability rather than using MVA. Tallent et al. [195] propose and evaluate three strategies to gain insight into the performance loss due to lock contention. Eyerman et al. [69], propose a probabilistic model to extend Amdahl's law to applications with critical sections. In addition, these effects are studied in the context of transactional databases [205, 101, 166, 9].

The models studied here represent a simple application with L locks, T threads and C cores as presented in Algorithm 1 on Page 72. In addition, the following assumptions about the program and the system are made for simplification:

- The number of threads are constant during the execution of the program, i.e., T is fixed. In addition, each thread runs on a dedicated core, i.e., $T = C$.
- All threads have the same lock access pattern, therefore t_{cs} and t_{ncs} are similar for all threads.

- The time it takes to acquire and release a lock does not change with increasing number of threads and is always 0. However, this does not hold in reality, and increasing number of threads affect the time it takes to hold the lock due to increased contention.
- It is assumed there are no additional overheads or delays in the system. In reality, there are additional overheads coming from scheduling or cache misses, but these are ignored in the model.

A probabilistic and an analytic model is used to demonstrate that the performance of an application with critical sections increases with increased concurrency up to a point beyond which the performance and utilization of the system does not increase by increasing concurrency.

6.1 Probabilistic Model

Eyerman et al. [69] propose an extension to Amdahl's law and show that parallel performance is not only limited by the sequential part of the program, but is also fundamentally limited by mutual exclusion. The extension to Amdahl's law is deduced by a probabilistic model that calculates the relative execution time of the program. The authors validate this model using synthetic simulation. This model is used in this section to find the maximum speedup and utilization of the system and the model is also evaluated by the benchmark described in Algorithm 1 (Page 72).

The sequential execution of program is split into three fractions. These fractions are the fractions of the time spent in each part of program normalized to the total time it takes for the program to run sequentially without parallelization and mutual exclusion. f_{seq} is the sequential fraction of the program's execution that cannot be parallelized. Further, $f_{par,ncs}$ is the fraction of the program that can be parallelized and does not need mutual exclusion, i.e., it's the non-critical section of the parallel fraction. Finally, $f_{par,cs}$ is the fraction of the program's execution that can be parallelized but is a critical section and thus requires mutual exclusion. The fractions sum up to one, i.e., $f_{seq} + f_{par,ncs} + f_{par,cs} = 1$.

The authors provide the following equation and show that the total execution time of a program with critical sections is proportional to:

$$Time \propto f_{seq} + max\left(f_{par,cs}P_{cs}P_{ctn} + \frac{f_{par,cs}(1 - P_{cs}P_{ctn}) + f_{par,ncs}}{T}; \right. \\ \left. f_{par,cs}P_{ctn} + \frac{f_{par,cs}(1 - P_{ctn}) + f_{par,ncs}}{2T} \right) \tag{6.1}$$

Where P_{ctn} is the *contention probability*, or the probability for *two* critical sections to contend, i.e., both critical sections are protected by the same lock. Also, P_{cs} is the probability for a critical section during parallel execution and is defined as:

$$P_{cs} = Pr\big[\text{critical section}\big|\text{parallel}\big] = \frac{f_{par,cs}}{f_{par,ncs} + f_{par,cs}} \tag{6.2}$$

Equation 6.1 is based on the fractions of execution time normalized to the sequential execution time. The expected value of the execution time of each iteration of the program can be found by replacing the fractions with the actual execution time of each section which yields the following:

$$\begin{aligned} Time(T) = t_{seq} + max\Bigg(& t_{par,cs}P_{cs}P_{ctn} + \frac{t_{par,cs}(1 - P_{cs}P_{ctn}) + t_{par,ncs}}{T}; \\ & t_{par,cs}P_{ctn} + \frac{t_{par,cs}(1 - P_{ctn}) + t_{par,ncs}}{2T}\Bigg) \end{aligned} \tag{6.3}$$

Next, integrating the parameters from Algorithm 1 yields the formula to model the expected time of each iteration. In the benchmark, there is no sequential part, i.e., $t_{seq} = 0$. In a single-threaded application, i.e., $T = 1$, there is no contention, hence $P_{ctn} = 0$. However, when $T > 1$ the probability for two threads contending with each other is the probability of one thread picking a lock and the second thread choosing the same lock, i.e., $P_{ctn} = 1/L$.

$$\begin{aligned} t_{seq} &= 0 \\ t_{par,cs} &= t_{cs} \\ t_{par,ncs} &= t_{ncs} \\ P_{ctn} &= \begin{cases} 0, & \text{if } T = 1 \\ \dfrac{1}{L}, & \text{otherwise} \end{cases} \\ P_{cs} &= \frac{f_{par,cs}}{f_{par,ncs} + f_{par,cs}} = \frac{t_{cs}}{t_{ncs} + t_{cs}} \end{aligned} \tag{6.4}$$

$$Time(T) = max\Bigg(\frac{t_{cs}^2}{L \cdot (t_{ncs} + t_{cs})} + \frac{t_{cs}(1 - \frac{t_{cs}}{L \cdot (t_{ncs} t_{cs})}) + t_{ncs}}{T}; \\ \frac{t_{cs}}{L} + \frac{t_{cs} \cdot (1 - \frac{1}{L}) + t_{ncs}}{2T} \Bigg) \tag{6.5}$$

The throughput of the system is defined to be the total number of iterations that all threads perform during the execution time of the program in each time unit. Throughput is denoted by $R(T)$ where T is the number of threads used to run the program. Since the right side of the Equation 6.5 approximates the time it takes to do a single iteration by each thread, the overall throughput for a unit time with T threads, $R(T)$, can be calculated as:

$$R(T) = \frac{1}{Time(T)} \tag{6.6}$$

This equation can be used to model the approximate throughput of the application. Furthermore, assuming there are an infinite number of threads in the system, i.e., $T \to \infty$, the maximum overall throughput of the program is calculated as:

$$\begin{aligned} \lim_{T \to \infty} R(T) &= \frac{1}{max\left(\frac{t_{cs}^2}{L \cdot (t_{ncs} + t_{cs})}; \frac{t_{cs}}{L} \right)} \\ &= \frac{L}{t_{cs}} \end{aligned} \tag{6.7}$$

Equation 6.7 indicates that the throughput of an application is limited by the number of locks and is inversely proportional to the duration of the critical section.

Utilization is defined as the number of cores that are being utilized after the system reaches steady state. The utilization is denoted by $U(T)$ where T is the number of threads used to run the program. By normalizing the throughput to the throughput of a single thread on a single core, which is $R(1) = 1/(t_{cs} + t_{ncs})$, the expected number of utilized cores can be obtained. Therefore, the expected utilization $U(T)$ of the application is:

$$U(T) = R(T) \cdot (t_{cs} + tncs) \tag{6.8}$$

The maximum utilization of the application is calculated as:

$$\lim_{T \to \infty} U(T) = \frac{L \cdot (t_{cs} + tncs)}{t_{cs}} \tag{6.9}$$

Equation 6.9 shows there is an upper bound to application speed-up when concurrency is increased.

6.2 Analytic Model

Pan et al. [152] provide an analytic model based on a closed queueing network. A queueing network consists of a set of nodes where customers travel between nodes where they receive a service. Each node can serve a single customer at a time (*single server node*) or multiple customers at the same time (*multiple server node*), or infinite number of customers (*delay centers*). When a customer arrives at a node with finite capacity (or *service center*) if the node is already serving the maximum number of customers, the new customer has to wait in a queue until one of the current customers leaves the node. However, if the node is a *delay center*, all customers receive service immediately without waiting.

In addition, each node has an average service time and costumers leave the node after the service time is passed. After customers are served at a node, they visit another node with a given probability. The flow of customers among nodes is defined by a routing matrix R, where R_{ij} represents the probability that a customer visits node j after visiting node i.

There are two types of queueing networks, open and closed networks. In open networks, customers constantly enter and leave the network at a certain rate and the network has a maximum capacity after which the customers that are trying to enter the system are rejected. However, a closed network has a constant number of customers and no customer leaves or enters the network. All customers in a closed queueing network circulate among different nodes in the network.

The program shown in Algorithm 1 can be modelled as a closed queueing network where the number of customers is equal to the number of threads in the system (T). Also, locks are represented by single server nodes, where service time corresponds to the time spent in the critical section (t_{cs}). The number of these single server nodes are equal to the number of locks (L). Moreover, the work done outside of the critical section is modelled as a visit to a *delay center* (i.e., *infinite-server node*) at which queueing is never needed. The service time (or delay) at the service center is equal to the time spent in the non-critical

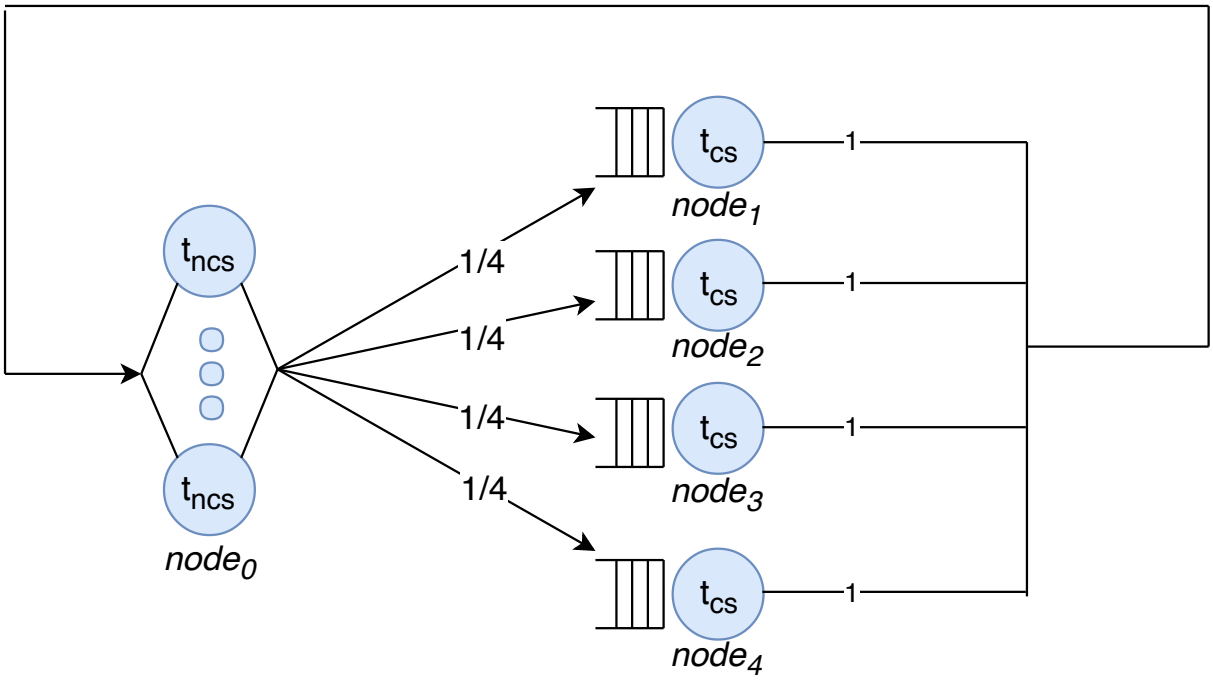


Figure 6.1: Closed queueing network for a program with 4 locks.

section (t_{ncs}). In addition, since locks are chosen randomly the probability that a customer visits a single-server node after visiting the delay center is $1/L$ and the probability of visiting the delay center after visiting one of the locked nodes is 1.

For instance, Figure 6.1 shows the closed queueing network network for an application with $L = 4$ locks and T threads. The probabilities are shown on the edges and service times are indicated on each node. Also, the routing matrix R is calculated as below if the order of nodes is $node_0, node_1, node_2, node_3$ and $node_4$:

$$R = \begin{bmatrix} 0 & \frac{1}{L} & \frac{1}{L} & \frac{1}{L} & \frac{1}{L} \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

To solve the closed queueing network above, MVA [162] is used, which gives an exact solution when service times are exponentially distributed. Pan et al. [152] model the lock

contention and lock times, which are the mean queue length and the mean queuing time in the one-server nodes. This model is used here to gain insight into the throughput and utilization of an application with critical-sections. Using a *class-dependent* queueing network, this model can be further extended to cover pipelined programs with multiple locks with various lock holding times and access orders.

As an example, the utilization for 16 critical sections and $t_{cs} = t_{ncs} = 1$ is shown in Figure 6.2. The utilization limit is

$$\lim_{n\to\infty} U(n) = \frac{L \cdot (t_{cs} + t_{ncs})}{t_{cs}} = 16 \times 2 = 32. \tag{6.10}$$

The graph shows both the probabilistic and the queueing-based model. These considerations have an important implication for the performance potential of multi-threaded software on multi-core hardware. When concurrent software must employ mutual exclusion, increasing the number of threads can potentially increase utilization and thus performance. It is also important to note that when the utilization provided by the model goes beyond the number of available cores on the system, there is no additional benefits as the application is limited by the number of cores on the system. The model is verified with a benchmark presented in the next Section.

6.2.1 Experimental Verification

In this section, the utilization model is evaluated using the benchmark described in Algorithm 1. Both work loops are set to 10μs (and 1ms in a second run). The number of locks is $L = 16$ and the number of available cores is $C = 32$. Figure 6.3 shows the number of work loops that are executed with an increasing number of system threads, normalized to the work of a single thread. The maximum CV is 0.001. Because the model does not include any overhead associated with low-level contention, context-switching, or scheduling, the maximum normalized work does not nearly reach 32 as predicted by the model, but only close to 13 cores. However, when repeating the experiment with a larger work loop (1ms), the overheads are more effectively masked and the normalized work converges to a higher number, more than 21, also shown in Figure 6.3. In both cases, the shape of the empirical curve resembles the shape of the models in Figure 6.2, which corroborates the basic conjecture.

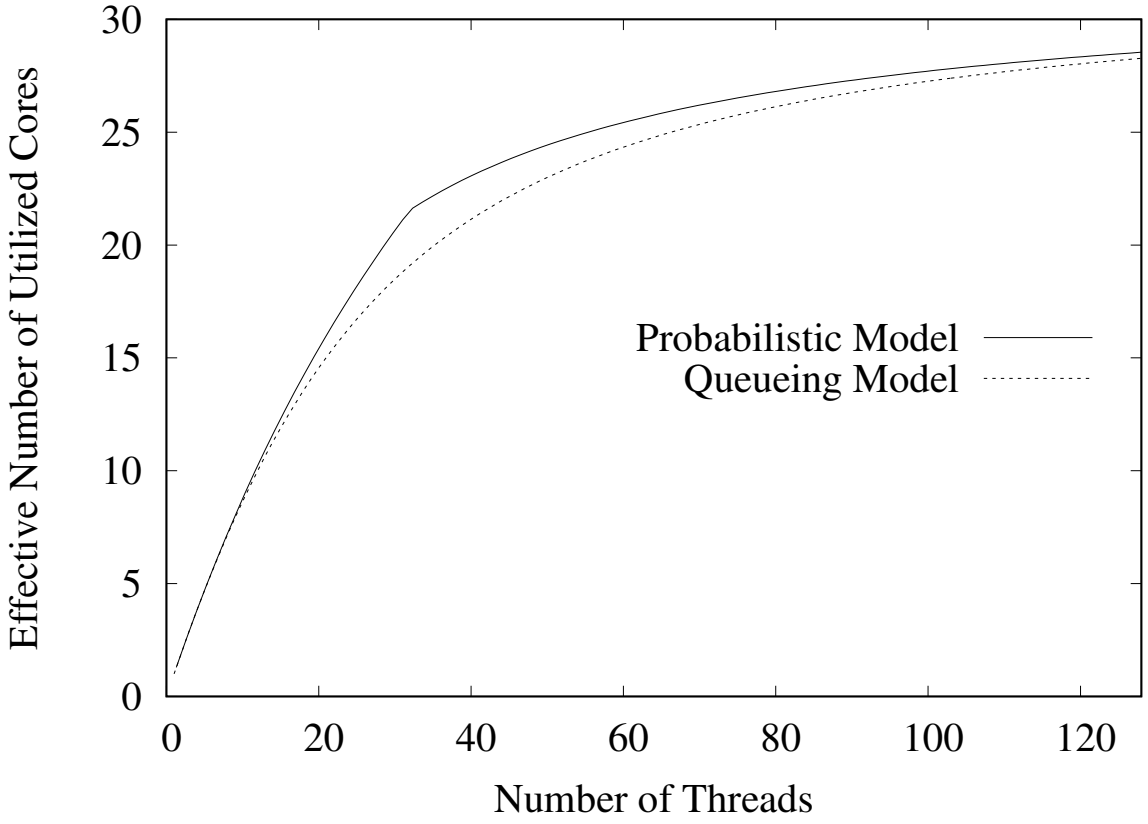


Figure 6.2: Utilization Models ($l = 16$, $c = w = 1$)

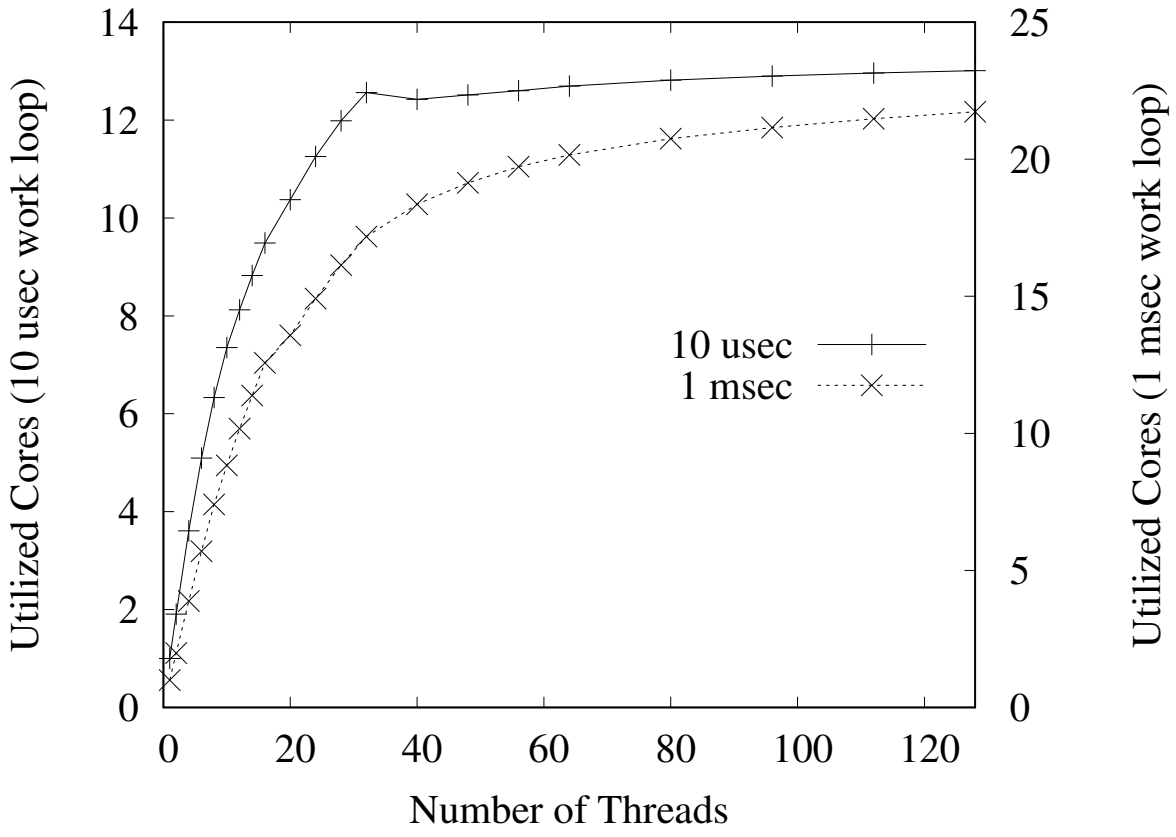


Figure 6.3: Model Verification (16 locks)

Chapter 7

Conclusion and Future Work

7.1 Summary and Conclusion

This thesis studies the performance of various user-level runtime systems and shows that it is possible to build an efficient and low-overhead user-level runtime system to handle large-scale concurrency. Scheduling and the I/O subsystem are identified as the primary building blocks of a user-level runtime, which affect the overall performance of the concurrent applications. A secondary building block is efficient locks to handle synchronization and mutual exclusion. Each of these components are studied in detail and multiple design alternatives for each component are presented.

In addition, a M:N user-level threading runtime is designed and implemented from scratch to study and evaluate the performance of these design alternatives. The goal is to create a bare-bone threading-library to present conflating language features with pure runtime-performance. The goal of this runtime is not to provide a comprehensive solution for all use cases of a user-level threading runtime. Thereby, the runtime benefits from a simple design and avoids the complexity and extra overhead of more sophisticated solutions, such as Go. By providing various compile-time flags to easily switch among different design alternatives, the runtime makes it possible to compare the performance of the design alternatives with each other.

The scheduler is a simple cooperative scheduler that supports work-sharing, work-pushing, and work-stealing strategies. The I/O subsystem is designed from scratch and provides a synchronous blocking interface to interact with I/O. Distributed and central polling mechanisms are presented for the I/O subsystem, and the frequency of polling is

shown to present a trade-off between overhead and latency. The synchronization infrastructure provides a blocking interface at the user-level, and a barging spectrum is presented that addresses approaches such as spinning, blocking, and baton passing.

The runtime is specifically designed to support large-scale network server applications that serve millions of concurrent connections. Each component of the runtime is evaluated individually using benchmarks and the results are compared with other M:N user-level threading runtimes, system-threads, and a best-in-class event-driven system. In addition, Memcached is used to perform a comprehensive evaluation of the runtime using a real-world application. However, Memcached uses an event-based model and therefore it is minimally transformed to use a thread-per-connection design that can operate with either kernel-level or user-level threads.

The results demonstrate that the fibre runtime presented here provides superior efficiency and performance compared to other user-level threading systems. This observation is not intended as an overall judgment of other systems, which typically provide advanced features not present in the fibre runtime. However, it does support the fundamental argument for this work – that a lean runtime is needed to fully understand the performance potential of user-level threading. The runtime is the first M:N user-level threading runtime that delivers this level of efficiency and scalability, comparable with a best-in-class event-based system.

In addition, it is shown that, contrary to popular belief, pthreads can handle a very large number of concurrent connections. The secondary conclusion from this work is that all runtime paradigms can deliver competitive performance – with some individual trade-offs. Most importantly, none of the runtime paradigms should be ruled out when assessing programming patterns for an application.

This thesis also extends the performance evaluation of the scheduler component to user-level actor runtime systems. An actor runtime has different requirements than a user-level threading runtime. Various characteristics of the actor model and message passing frameworks are investigated that can affect execution performance on NUMA machines. It is conjectured that actors can benefit from locality-aware schedulers and a study of the effectiveness of locality-aware schedulers for actor runtime systems is presented. In addition, the applicability of existing work-stealing schedulers is discussed. These findings are used to develop two variants of a novel locality-aware work stealing scheduler for the C++ Actor Framework (CAF) runtime that takes into account the distance between cores and NUMA nodes. The performance of these schedulers is compared with CAF's default randomized victim scheduler. Locality-aware work stealing shows comparable or better performance in most cases. However, it is also demonstrated that the effectiveness of

locality-aware schedulers depends on the workload.

In addition, two synchronization models are studied to understand the impact of concurrency on applications. These models are used to estimate the maximum utilization and throughput of applications that have critical sections. It is shown that without any overhead, increasing concurrency can lead to better performance depending on the number of locks, lock holding time, and the duration of the non-critical section of the application. However, in reality a runtime system and synchronization add significant overhead, and this subject should be further studied using real-world applications to understand the impact on concurrency.

7.2 Future Work

This thesis attempts to provide an exhaustive evaluation of the user-level threading runtime using benchmarks and applications. However, some follow-up investigations can be done as complementary work. For instance, evaluating the capacity and behaviour of the runtime in handling very large number of connections requires further investigations. All experiments in this thesis are executed locally on a multi-core computer partitioned to run both client and server to eliminate the cost of device communication. Therefore, evaluating the performance of the runtime using real network interactions is another interesting direction for future work.

Furthermore, this thesis evaluates user-level runtimes using specific concurrent applications, for instance user-level threading is evaluated using network servers, while the actor runtime is evaluated using communication patterns that exist in actor applications. However, there is an overlap between the functionalities of different user-level runtimes. For instance, a work-stealing scheduler is used for HPC and also for network servers for better resource utilization. But each application type has different requirements, and the question is, to what extent these runtimes are similar in terms of scheduling requirements. Thus, a further subject for research is to extend the user-level threading runtime presented in this thesis for HPC workloads and to investigate whether the I/O component can satisfy HPC requirements. Moreover, the actor model has always been studied separately from HPC workloads; hence, another area for further research is to study the similarities between actor runtimes and HPC workloads and investigate whether the actor model can be applied to address CPU bound HPC computations.

In the long run, this work might hint at a new division of work between kernel and user-level. The preemptive kernel scheduler could focus on resource management and fairness

between applications, while a multitude of modular user-level schedulers would be geared towards the specific needs of different types of applications. In addition, the current polling subsystem inside the kernel in Linux and FreeBSD have some shortcomings that lead to unnecessary overhead when interacting with user-level threading runtimes. The findings in this thesis can be used towards designing a more efficient polling infrastructure that avoids these sources of overhead.

References

[1] epoll - i/o event notification facility. http://man7.org/linux/man-pages/man7/epoll.7.html.

[2] libuv: Asynchronous i/o made simple. https://libuv.org, 2017.

[3] The rust programming language. http://www.rust-lang.org/, 2017.

[4] Select is fundamentally broken. https://idea.popcount.org/2017-01-06-select-is-fundamentally-broken, 2017.

[5] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[6] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management without Manual Stack Management. In *Proceedings of USENIX 2002 Annual Technical Conference*. USENIX, 2002.

[7] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[8] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, September 1990.

[9] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, nov 1987.

[10] George Almasi. Pgas (partitioned global address space) languages. In *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer, 2011.

[11] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations. *ACM SIGOPS Operating Systems Review*, 25(5):95–109, oct 1991.

[12] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[13] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 33–42, New York, NY, USA, 2012. ACM.

[14] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Meetings Jim Demmel, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The landscape of parallel computing research: A view from berkeley.

[15] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[16] Islam Atta, Pinar Tozun, Anastasia Ailamaki, and Andreas Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 188–198, Washington, DC, USA, 2012. IEEE Computer Society.

[17] Islam Atta, Pinar Tzn, Anastasia Ailamaki, and Andreas Moshovos. Reducing OLTP Instruction Misses with Thread Migration. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 9–15, New York, NY, USA, 2012. ACM.

[18] Islam Atta, Pinar Tzn, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. STREX: Boosting Instruction Cache Reuse in OLTP Workloads Through Stratified Transaction Execution. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 273–284, New York, NY, USA, 2013. ACM.

[19] Paul A. Bailes. Introduction to functional programming. *Science of Computer Programming*, 12(2):158–164, jul 1989.

[20] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 19–19, Berkeley, CA, USA, 1999. USENIX Association.

[21] Blaise Barney and Lawrence Livermore. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads/, 2017.

[22] Jason Baron. epoll: add epollexclusive flag. https://lwn.net/Articles/667087, 2015.

[23] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, mar 2017.

[24] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. RaftLib. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '15*. ACM Press, 2015.

[25] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. *Microsoft Research*, March 2014.

[26] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[27] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[28] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[29] OAR Board. Openmp application program interface version 3.0. In *The OpenMP Forum, Tech. Rep*, 2008.

[30] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21(2):246–254, may 1994.

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

[32] Jonas Bonér. Introducing akka - simpler scalability, fault-tolerance, concurrency & remoting through actors, 2010.

[33] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Edited R. M. Ramanathan, Vince Thomas, Intel Corporation, and Stephen S. Pawlowski. Intel processor and platform evolution for the next decade executive summary, 2017.

[34] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[35] Tim Brecht. The μserver project. https://userver.uwaterloo.ca/, 2015.

[36] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.

[37] Neil Brown. C++csp2: Easy concurrency for c++. https://www.cs.kent.ac.uk/projects/ofa/c++csp/, 2017.

[38] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. *μC++: Concurrency in the Object-Oriented Language C++*. 1992.

[39] Peter A. Buhr, David Dice, and Wim H. Hesselink. High-performance N-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, March 2015.

[40] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of CCGRID*, pages 521–530, 2006.

[41] Alan Burns. *Programming in OCCAM 2*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[42] Adrian Castello, Antonio J. Pena, Sangmin Seo, Rafael Mayo, Pavan Balaji, and Enrique S. Quintana-Orti. A review of lightweight thread approaches for high performance computing. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, sep 2016.

[43] Barcelona Supercomputing Center. Nanos++. https://pm.bsc.es/projects/nanox/, 2017.

[44] Barcelona Supercomputing Center. The ompss programming model. https://pm.bsc.es/ompss/, 2017.

[45] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. *SIGOPS Oper. Syst. Rev.*, 40(5):283–292, October 2006.

[46] Dominik Charousset, Raphael Hiesgen, and Thomas C Schmidt. Caf-the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 15–28. ACM, 2014.

[47] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, April 2016.

[48] Quan Chen, Minyi Guo, and Zhiyi Huang. Adaptive cache aware bitier work-stealing in multisocket multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2334–2343, 2013.

[49] Quan Chen, Zhiyi Huang, Minyi Guo, and Jingyu Zhou. Cab: Cache aware bi-tier task-stealing in multi-socket multi-core architecture. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 722–732. IEEE, 2011.

[50] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE! 2015*. ACM Press, 2015.

[51] Alexander Collins, Christian Fensch, and Hugh Leather. Auto-tuning parallel skeletons. *Parallel Processing Letters*, 22(02), 2012.

[52] Oracle Corporation. Programming with solaris threads. https://docs.oracle.com/cd/E19455-01/806-5257/6je9h033n/index.html, 2017.

[53] Isaac Councill. khttpd linux http accelerator. http://www.fenrus.demon.nl, 2017.

[54] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[55] Hoang-Vu Dang, Marc Snir, and William Gropp. Towards millions of communicating threads. In *Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016*. ACM Press, 2016.

[56] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[57] Beman Dawes, David Abrahams, and Rene Rivera. Boost c++ libraries. http://www.boost.org/, 2017.

[58] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 31–40, New York, NY, USA, 2016. ACM.

[59] J. del Cuvillo, Weirong Zhu, Ziang Hu, and G.R. Gao. TiNy threads: A thread virtual machine for the cyclops64 cellular architecture. In *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005.

[60] A Micro Devices. Amd64 architecture programmers manual volume 2: System programming, 2006.

[61] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[62] Dormando. Memcached: A distributed memory object caching system. http://memcached.org, 2017.

[63] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Transactions on Architecture and Code Optimization*, 11(3):1–25, aug 2014.

[64] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable Task Parallelism for NUMA. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT'16*. ACM Press, 2016.

[65] Ulrich Drepper. What every programmer should know about memory, 2007.

[66] Akos Dudás, S Juhász, and S Kolumbán. Performance analysis of multi-threaded locking in bucket hash tables. In *ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica*, volume 36, pages 63–74, 2012.

[67] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[68] Ralf S. Engelschall. Gnu pth - the gnu portable threads. http://www.gnu.org/software/pth/, 2017.

[69] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. *ACM SIGARCH Computer Architecture News*, 38(3):362, jun 2010.

[70] Michael J Flynn and Kevin W Rudd. Parallel architectures. *ACM Computing Surveys (CSUR)*, 28(1):67–70, 1996.

[71] The Apache Software Foundation. Apache web server. http://httpd.apache.org/, 2017.

[72] E. Francesquini, A. Goldman, and J. F. Mhaut. A NUMA-Aware Runtime Environment for the Actor Model. In *2013 42nd International Conference on Parallel Processing*, pages 250–259, October 2013.

[73] Emilio Francesquini, Alfredo Goldman, and Jean-Franois Mhaut. Actor Scheduling for Multicore Hierarchical Memory Platforms. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 51–62, New York, NY, USA, 2013. ACM.

[74] Emilio Francesquini, Alfredo Goldman, and Jean-Franois Mhaut. Improving the Performance of Actor Model Runtime Environments on Multicore and Manycore Platforms. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, pages 109–114, New York, NY, USA, 2013. ACM.

[75] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Linux Symposium*, volume 1, 2004.

[76] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA systems. *Communications of the ACM*, 58(12):59–66, nov 2015.

[77] Will Glozer. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. [Online; accessed 2018-05-01].

[78] Lu Gong, Zhanqiang Li, Tao Dong, and Youkai Sun. Rethink scalable m:n threading on modern operating systems. *Journal of Computers*, 11(3):176–188, may 2016.

[79] Google. The go programming language. https://golang.org/, 2017.

[80] Richard Graham, Galen Shipman, Brian Barrett, Ralph Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006.

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

[82] Dominik Grewe and Michael FP OBoyle. A static task partitioning approach for heterogeneous systems using opencl. In *Compiler Construction*, pages 286–305. Springer, 2011.

[83] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[84] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[85] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[86] Andreas Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, 2005.

[87] Christian Gyrling. Parallelizing the naughty dog engine using fibers. http://www.gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine, 2015. This talk is a detailed walkthrough of the game engine modifications needed to make The Last of Us Remastered run at 60 fps on PlayStation 4. Topics covered will include the fiber-based job system Naughty Dog adopted for the game, the overall frame-centric engine design, the memory allocation patterns used in the title, and our strategies for dealing with locks.

[88] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2015.

[89] Ashif Harji. Performance comparison of uniprocessor and multiprocessor web server architectures. https://uwspace.uwaterloo.ca/handle/10012/5040.

[90] Haskell. Servant. https://haskell-servant.github.io/, 2018.

[91] haskell.org. Haskell. https://www.haskell.org, 2018.

[92] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[93] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[94] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[95] C. Hewitt and H. G. Baker. ACTORS AND CONTINUOUS FUNCTIONALS. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[96] Carl Hewitt, Peter Bishop, and Richard Steiger. Artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

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

[98] Shams Imam and Vivek Sarkar. Savina-an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

[99] Typesafe Inc. Akka. http://akka.io/, 2017.

[100] Intel Corporation. Intel cilk plus. https://www.cilkplus.org, 2018.

[101] Keki B. Irani and Hing-Lung Lin. Queueing network models for concurrent transaction processing in a database system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data - SIGMOD '79*. ACM Press, 1979.

[102] E Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and K Park. mtcp: a highly scalable user-level tcp stack for multicore systems. *Proc. 11th USENIX NSDI*, 2014.

[103] Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. A new look at the roles of spinning and blocking. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware - DaMoN '09*. ACM Press, 2009.

[104] M Tim Jones. Boost application performance using asynchronous i/o. *IBM developerWorks*, 2006.

[105] L. V. Kalé, J. Yelon, and T. Knauff. Threads for interoperable parallel programming. In *Languages and Compilers for Parallel Computing*, pages 534–552. Springer Berlin Heidelberg, 1997.

[106] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications - OOPSLA '93*. ACM Press, 1993.

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

[108] Richard Kettlewell. poll() and eof. http://www.greenend.org.uk/rjk/tech/poll.html.

[109] Yoongu Kim, Dongsu Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[110] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[111] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, mar 2005.

[112] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[113] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*. ACM Press, 2000.

[114] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[115] Marc Lehmann. libeio. http://software.schmorp.de/pkg/libeio.html, 2017.

[116] Marc Lehmann and Emanuele Giaquinta. libev. http://software.schmorp.de/pkg/libev.html, 2017.

[117] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC '09*. ACM Press, 2009.

[118] Jonathan Lemon. Kqueue - a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153, Berkeley, CA, USA, 2001. USENIX Association.

[119] Jacob Leverich. Mutilate. https://github.com/leverich/mutilate. [Online; accessed 2018-05-01].

[120] Christian Bienia Kai Li. Characteristics of workloads using the pipeline programming model. In Ana Lucia Varbanescu, Anca Molnos, and Rob van Nieuwpoort, editors, *Computer Architecture*, number 6161 in Lecture Notes in Computer Science, pages 161–171. Springer Berlin Heidelberg, 2011.

[121] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *ACM SIGPLAN Notices*, 42(6):189, jun 2007.

[122] 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 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 189–199, New York, NY, USA, 2007. ACM.

[123] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[124] Davide Libenzi. Improving (network) i/o performance. http://www.xmailserver.org/linux-patches/nio-improve.html, 2002.

[125] lighttpd. weighttp - a lightweight and simple webserver benchmarking tool. https://github.com/lighttpd/weighttp, 2018.

[126] Jean-Pierre Lozi, Florian David, Gal Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[127] Paul Mackay. Why has the actor model not succeeded. *Surveys and Presentations in Information Systems Engineering, London*, 1997.

[128] Kostas Magoutis, Margo Seltzer, and Eran Gabber. The case against user-level networking. In *Third Workshop on Novel Uses of System Area Networks (SAN-3)(Held in conjunction with HPCA-10)*, 2004.

[129] Zoltan Majo and Thomas R. Gross. Memory management in NUMA multicore systems. *ACM SIGPLAN Notices*, 46(11):11, nov 2011.

[130] Zoltan Majo and Thomas R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[131] John L Manferdelli, Naga K Govindaraju, and Chris Crall. Challenges and Opportunities in Many-Core Computing. *Proceedings of the IEEE*, 96(5):808–815, May 2008.

[132] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *ACM SIGOPS Operating Systems Review*, 25(5):110–121, oct 1991.

[133] Nick Mathewson, Azat Khuzhin, and Niels Provos. Libevent - and event notification library. http://libevent.org/, 2017.

[134] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Department of Computer Science School of Engineering and Applied Science, 2017.

[135] Dave McCracken. Posix threads and the linux kernel. In *Ottawa Linux Symposium*, page 330, 2002.

[136] Jiayuan Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *IEEE International Conference on Computer Design, 2009. ICCD 2009*, pages 282–288, 2009.

[137] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166. ACM. ACM ID: 1755930.

[138] Microsoft. Fibres. https://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx, 2017.

[139] Microsoft. Windows user-mode scheduling. https://msdn.microsoft.com/en-us/library/windows/desktop/dd627187(v=vs.85).aspx, 2017.

[140] Seung-jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[141] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society.

[142] Mozy Inc. Mordor: a high performance i/o library based on fibers. https://github.com/mozy/mordor, 2008.

[143] Jun Nakashima and Kenjiro Taura. MassiveThreads: A thread library for high productivity languages. In *Lecture Notes in Computer Science*, pages 222–238. Springer Berlin Heidelberg, 2014.

[144] ntop. Pf ring: High-speed packet capture, filtering and analysis. http://www.ntop.org/products/pf_ring/, 2017.

[145] OCaml Labs. Multicore ocaml. http://ocamllabs.io/doc/multicore.html, 2018.

[146] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, and Jan F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 49–56, New York, NY, USA, 2011. ACM.

[147] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124, 2012.

[148] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, September 1996.

[149] David Ott. Optimizing applications for NUMA. https://software.intel.com/en-us/articles/optimizing-applications-for-numa, 2017.

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

[151] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. *ACM SIGPLAN Notices*, 45(6):376, may 2010.

[152] Xiaoyue Pan, Jonatan Lindn, and Bengt Jonsson. Predicting the Cost of Lock Contention in Parallel Applications on Multicores using Analytic Modeling. In *DIVA*, 2012.

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

[154] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A unified parallel runtime for clusters of NUMA machines. In *Lecture Notes in Computer Science*, pages 78–88. Springer Berlin Heidelberg.

[155] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[156] Laercio L. Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Chao Mei, Abhinav Bhatele, Philippe O. A. Navaux, Francois Broquedis, Jean-Francois Mehaut, and Laxmikant V. Kale. A hierarchical approach for load balancing on parallel multi-core systems. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP '12, pages 118–127, Washington, DC, USA, 2012. IEEE Computer Society.

[157] Antoniu Pop and Albert Cohen. OpenStream. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–25, jan 2013.

[158] Allan Porterfield, Nassib Nassar, and Rob Fowler. Multi-threaded library for many-core systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, may 2009.

[159] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas D. Jordan. Proactor - an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events.

[160] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. Multi-core and network aware mpi topology functions. In *European MPI Users' Group Meeting*, pages 50–60. Springer, 2011.

[161] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008(173), September 2008.

[162] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM*, 27(2):313–322, apr 1980.

[163] Didier Rémy. Using, understanding, and unraveling the ocaml language from practice to theory and vice versa. In *Applied Semantics*, pages 413–536. Springer, 2002.

[164] Riot Games. Chat Service Architecture: Servers. https://engineering.riotgames.com/news/chat-service-architecture-servers, accessed 2018-01-02.

[165] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204. ACM, 2008.

[166] In Kyung Ryu and Alexander Thomasian. Analysis of database performance with dynamic locking. *Journal of the ACM*, 37(3):491–523, jul 1990.

[167] Andreas Sandberg, David Eklv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society.

[168] Douglas C. Schmidt. *Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching.*

[169] Colin Scott. Latency numbers every programmer should know. http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html, 2017.

[170] ScyllaDB. Seastar. http://www.seastar-project.org/, 2017.

[171] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrian Castello, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Sanjay Kale, SRIRAM KRISHNAMOORTHY, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2017.

[172] Chris Siebenmann. Bsd unix developed over more time than i usually think. https://utcc.utoronto.ca/~cks/space/blog/unix/BSDExtendedDevelopment, 2015.

[173] Inc Silicon Graphics. State threads for internet applications. http://state-threads.sourceforge.net/docs/st.html, 2015.

[174] Oliver Sinnen. *Task Scheduling for Parallel Systems.* John Wiley & Sons, Inc., apr 2007.

[175] David B Skillicorn. *Foundations of parallel programming.* Number 6. Cambridge University Press, 2005.

[176] Michael Snoyman. Yesod web framework. https://www.yesodweb.com, 2018.

[177] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[178] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. Apple Academic Press Inc., 2015.

[179] Facebook Open Source. Folly: Facebook open-source library. https://github.com/facebook/folly, 2017.

[180] Facebook Open Source. Rocksdb: A persistance key-value store. http://rocksdb.org/, 2017.

[181] Priyaa Varshinee Srinivasan. Improving data locality in applications using message passing.

[182] Stackless. Stackless python. https://bitbucket.org/stackless-dev/stackless/wiki/Home, 2017.

[183] stefano casazza. Ulib. https://github.com/stefanocasazza/ULib, 2018.

[184] DANIEL STENBERG. Wsapoll is broken. https://daniel.haxx.se/blog/2012/10/10/wsapoll-is-broken, 2012.

[185] DANIEL STENBERG. Poll on mac 10.12 is broken. https://daniel.haxx.se/blog/2016/10/11/poll-on-mac-10-12-is-broken/, 2016.

[186] Thomas Sterling, Matthew Anderson, P. Kevin Bohan, Maciej Brodowicz, Abhishek Kulkarni, and Bo Zhang. Towards exascale co-design in a runtime system. In *Exascale Applications and Software Conference*, Stockholm, Sweden, Apr 2014.

[187] W Richard Stevens, Bill Fenner, and Andrew M Rudoff. *UNIX network programming*, volume 1. Addison-Wesley Professional, 2004.

[188] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. On the efficiency of localized work stealing. *Information Processing Letters*, 116(2):100 – 106, 2016.

[189] Sun Microsystems, Inc. Multithreading in the solaris operating environment. Technical report, Sun Microsystems, Inc., 2002.

[190] Martin Sustrik. libdill: Structured concurrency for c. http://libdill.org/, 2017.

[191] Martin Sustrik. Libmill: Go-style concurrency in c. http://libmill.org/, 2017.

[192] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm, 2017.

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

[194] Alexander Szlavik. Cache-aware virtual page management. 2013.

[195] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. *ACM SIGPLAN Notices*, 45(5):269, may 2010.

[196] David Tam, Reza Azimi, and Michael Stumm. Thread clustering. *ACM SIGOPS Operating Systems Review*, 41(3):47, jun 2007.

[197] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP 2013 – Object-Oriented Programming*, pages 302–326. Springer Berlin Heidelberg, 2013.

[198] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '99*. ACM Press, 1999.

[199] Kenjiro Taura and Akinori Yonezawa. Fine-grain multithreading with minimal compiler support—a cost effective approach to implementing efficient multithreading languages. *ACM SIGPLAN Notices*, 32(5):320–333, may 1997.

[200] Ian Lance Taylor. Split stacks in gcc. https://gcc.gnu.org/wiki/SplitStacks, 2018.

[201] TechEmpower. Techempower benchmarks. https://www.techempower.com/benchmarks, 2018.

[202] Tencent. Libco. https://github.com/Tencent/libco, 2017.

[203] The NetBSD Foundations. Significant changes from netbsd 4.0 to 5.0, 2009.

[204] Samuel Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, United States, June 2005. ICS / ACM / IRISA.

[205] A. Thomasian. On a More Realistic Lock Contention Model and Its Analysis. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, pages 2–9, Feb 1994.

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

[207] M. O. Tokhi, M. A. Hossain, and M. H. Shaheed. *Parallel Computing for Real-time Signal Processing and Control*. Springer London, 2003.

[208] Parallel Universe. Quasar: lightweight threads and actors for the jvm. http://www.paralleluniverse.co/quasar/, 2017.

[209] U. Vahalia. *Unix Internals: The New Frontiers*. Pearson Education, 1996.

[210] Aliaksandr Valialkin. Fast HTTP package for Go. https://github.com/valyala/fasthttp. [Online; accessed 2018-05-01].

[211] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, December 2001.

[212] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

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

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

[215] Zeljko Vrba, Havard Espeland, Pal Halvorsen, and Carsten Griwodz. Limits of Work-Stealing Scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 280–299. Springer, Berlin, Heidelberg, May 2009.

[216] Zeljko Vrba, Pal Halvorsen, and Carsten Griwodz. A simple improvement of the work-stealing scheduling algorithm. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '10, pages 925–930, Washington, DC, USA, 2010. IEEE Computer Society.

[217] Dmitry Vyukov. Intrusive mpsc node-based queue. http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue, 2014.

[218] V.M. Weaver. Linux perf event features and overhead. 01 2013.

[219] Matt Welsh. Volatile and decentralized: A retrospective on SEDA, 2015.

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

[221] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, apr 2008.

[222] Sebastian Wölke, Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. Locality-guided scheduling in caf. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, pages 11–20, New York, NY, USA, 2017. ACM.

[223] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[224] Jixiang Yang and Qingbi He. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming*, pages 1–25, 2017.

[225] George Yunaev. select / poll / epoll: practical difference for system architects. https://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects/, 2014.

[226] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazieres, and M Frans Kaashoek. Multiprocessor support for event-driven programs. In *USENIX 2003 Annual Technical Conference*, pages 239–252. USENIX, 2003.

[227] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142. ACM, 2010.

[228] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.

APPENDICES

Appendix A

Other User-level Threading Libraries

MAESTRO [158] provides lightweight threads and synchronization among them. It is designed to be the target of a high-level language compiler or source-to-source translator, not for user-level programming. MAESTRO is addressing performance in HPC frameworks and separates the size of the hardware being used from the amount of parallelism available in the application. Its runtime uses one thread per node to monitor dynamic application and hardware state to provide flexible and dynamic scheduling. The monitor thread interacts with the scheduler and the operating system and observes hardware counters to determine the local performance. Based on the local and global observations, MAESTRO communicates with runtime schedulers to recommend different configurations and change the number of working threads. MAESTRO is not a user-level threading library, and is only intended to support programming environments and compilers to create the amount of parallelism and synchronization needed in a simple and efficient manner.

Dang et al. [55] present a Message Passing Interface (MPI) runtime design that uses user-level threads to lower the overhead of MPI applications. They introduce a light weight thread scheduler using a bit-vector that requires a single write for marking a thread as runnable. Also, they exploit a hash table to support a constant time overhead algorithm for MPI point-to-point communication. Locally, they use a scheduler that understands synchronization objects, and the Network Interface Controller (NIC) can be accessed in user space. Nonetheless, this work is mainly trying to optimize the communication among user-level threads and lower the overhead.

Converse [105] is a thread subsystem that supports both threads and stackless threads (i.e., tasklets) and also supports interoperability among parallel programming languages. Converse provides a customizable schedulers where programmers can implement their own

schedulers for different part of an application. It also supports message passing to share data using Message Passing Interface (MPI). Converse provides a complete framework for implementing inter-operable multi-threaded programming languages rather than being a independent threading runtime. For example, Charm++ [106] is implemented on top of Converse to run tasklets and user-level threads. Converse is the motivation for Argobots, but Argobots delivers more flexible and deterministic threading and tasking primitives by allowing users to control every detail of Argobots.

HPX-5 [186] uses user-level threads that has unified access to a high-performance Partitioned Global Address Space (PGAS) for fine-grained execution of tasks. It supports scalable distributed scientific applications using PGAS.

Castelló et al. [42] designed a set of benchmarks to evaluate lightweight thread approaches for high performance computing. They evaluate and compare MassiveThreads [143], Qthreads [221], Argobots [171], and Go [79] and show that Argobots, Qthreads, and MassiveThreads outperform other approaches (including OpenMP [54]).

C++ CSP2 bring CSP to C++ in both Linux and Windows using M:N mapping and cooperative scheduling. User-level threads can communicate using channels and each thread has a fixed-size stack. However, the programmer has to do the load balancing manually through running processes (i.e., user-level threads) in parallel or in sequence over current kernel thread or a new kernel thread. The process that starts other processes, pauses execution and waits for other processes to finish. This work is mainly focused on communication among processes and does not provide scheduling or I/O support, and the parallelization of tasks is limited.

Marcel [204] uses user-level thread to provide hierarchical scheduling on NUMA machines. Protothreads [67] is focused on tasklets (i.e., lightweight, stackless threads) and is designed for memory constrained systems such as small embedded systems or wireless sensor network nodes. It provides linear code execution for event-driven systems implemented in C. Stackless python [182] also focuses on tasklets in the python language.

Cooperative fibres are also used in a small scale to parallelize the workload and improve CPU utilization in the Naughty Dog game engine [87]. TiNy threads [59] is specialized to map lightweight software threads to hardware-thread units in the Cyclops64 cellular architecture. Nanos++ [43] provides user-level threads that are used to implement task parallelism using synchronization based on data-dependencies in OmpSs (an extension to the OpenMP programming model) [44]. The main goal of this framework is for research of parallel programming environments. Quasar [208] is an attempt to bring green threads to Java by using exceptions and bytecode rewriting to handle continuations.

Lithe [151] avoids resource over-subscription by providing nonvirtualized processing

resources which are shared cooperatively by the application libraries. Lithe provides two primitives, *harts* and *contexts*, to enable library runtime to perform parallel execution. *harts* are threads that are allocated by Lithe and assigned to applications and prevent over-subscription by having one *hart* per core. *contexts* are user-level threads that act as the execution vessel for the computation running on the hart. Applications are responsible for implementation of their own cooperative scheduler to schedule contexts on the harts under their control. Lithe runtime keeps track of the hierarchy of schedulers. Each scheduler can request more harts, or if not required, release the harts to the runtime. Application libraries should be ported and linked against Lithe runtime to share harts and be able to run their own contexts.

Seastar [170] is a C++ framework based on message passing and shared-nothing design. The concurrency support in this framework comes from futures and promises. Seastar moves the network stack fully into user-space to avoid context switches, interrupts and threaded model that exists in the kernel-level network-stack.

Appendix B

Results From The Intel Machine

The figures presented in this appendix are the results of the experiments in Chapter 4 executed on an Intel machine. Each figure corresponds to a figure in Chapter 4.

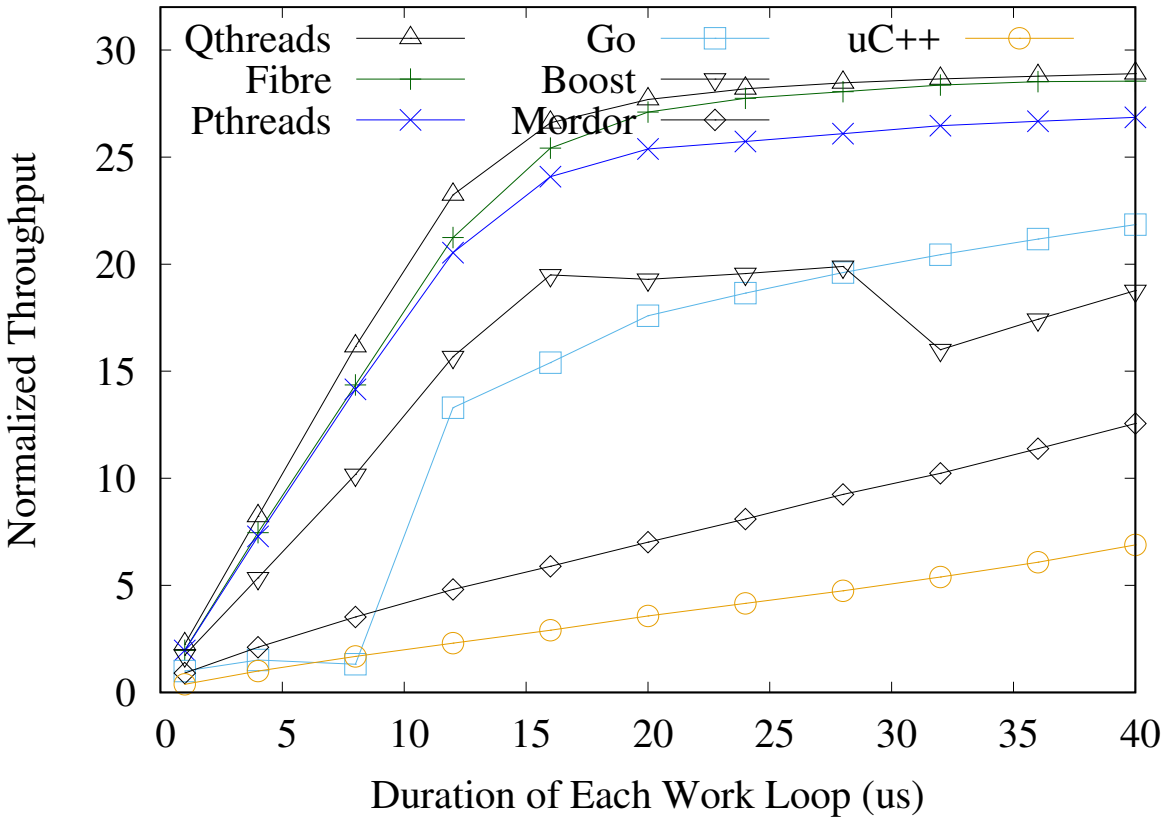


Figure B.1: Scalability (32 cores, 1024 fibres, 32 locks) on Intel, Corresponds to Figure 4.5

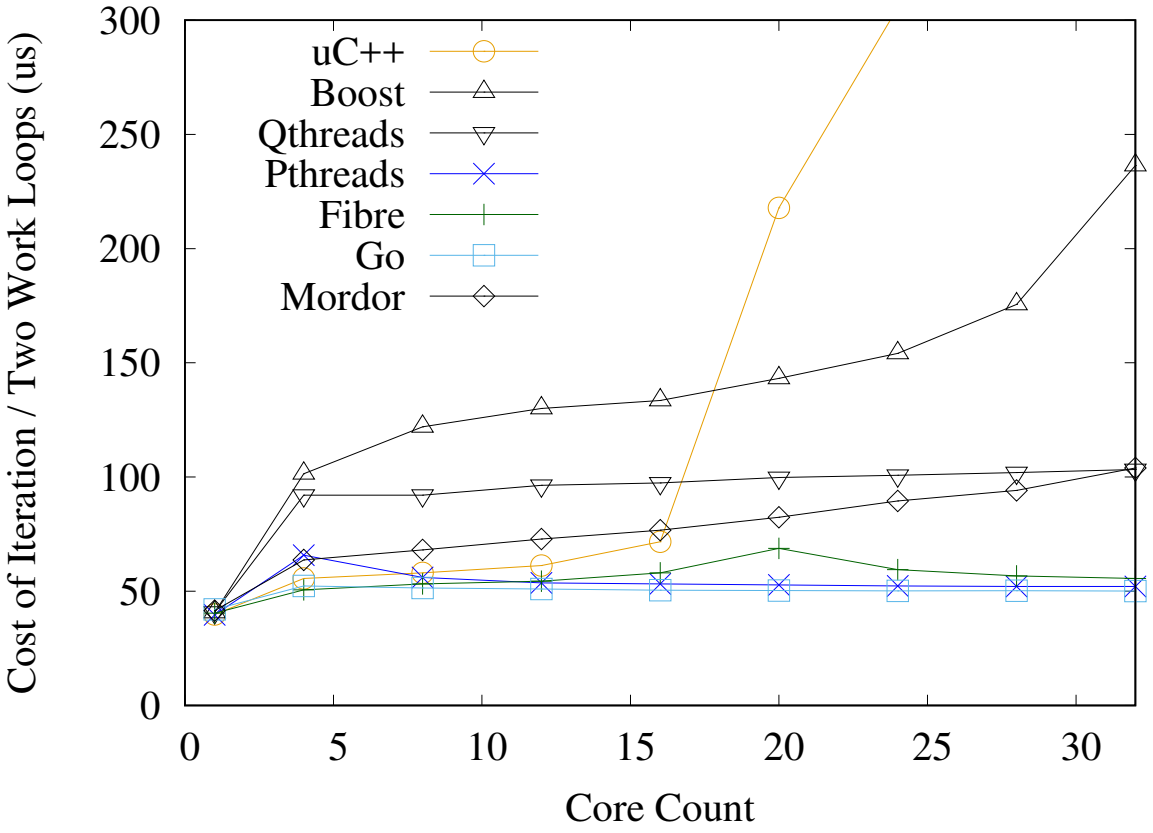


Figure B.2: Efficiency (N^2 fibres, $N/4$ locks, 20μs loops) on Intel, Corresponds to Figure 4.6

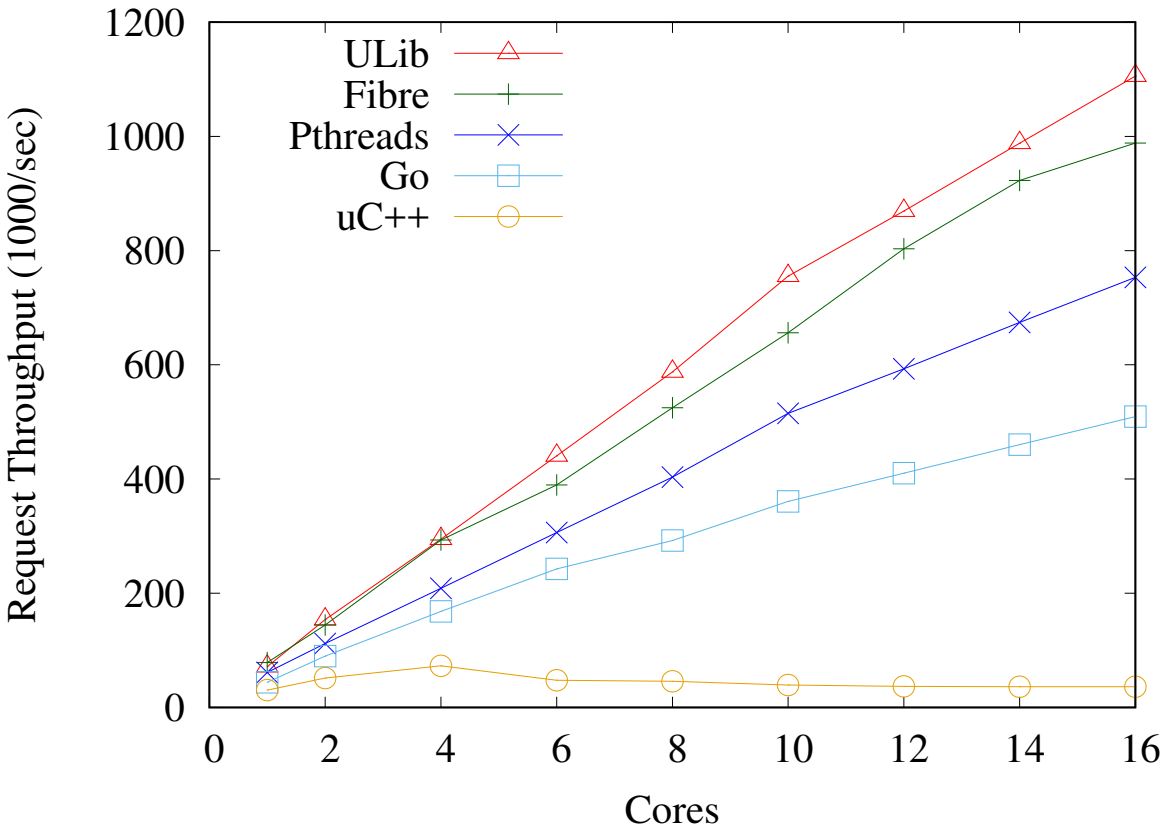


Figure B.3: Web Server (12,000 connections) on Intel, Corresponds to Figure 4.7

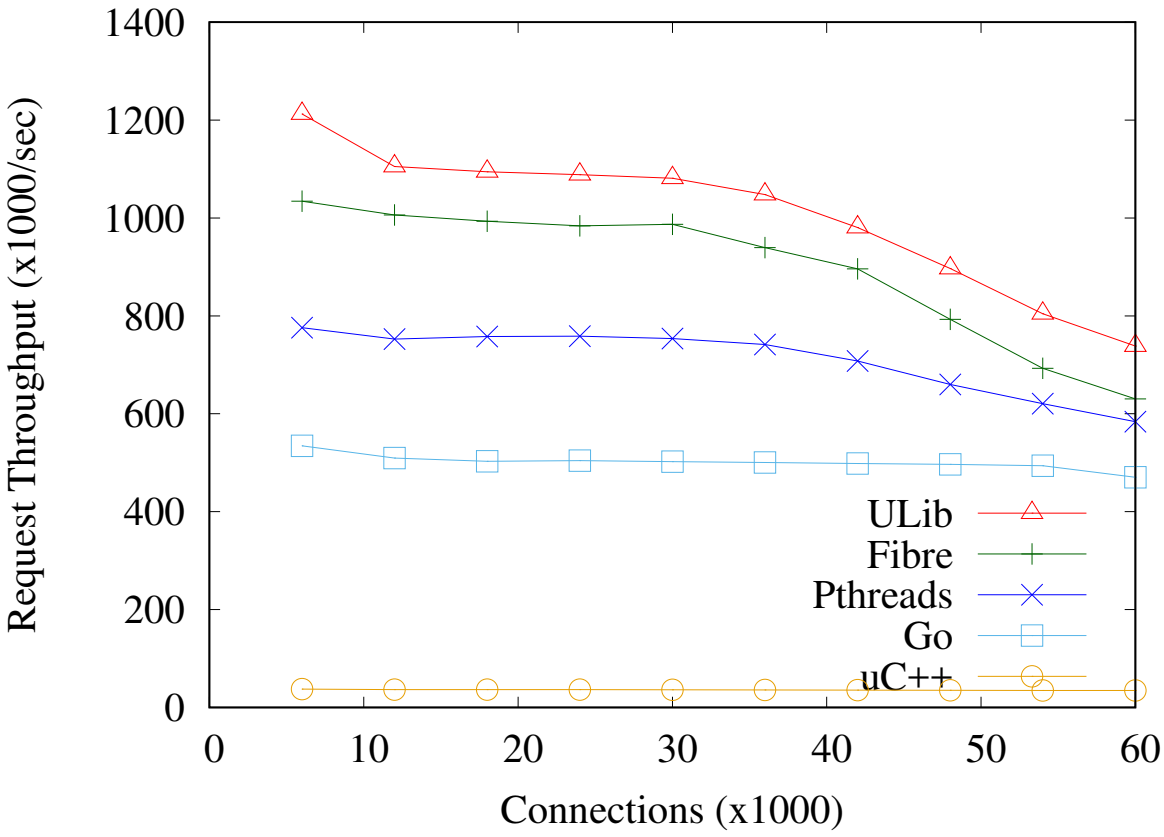


Figure B.4: Web Server (16 cores) on Intel, Corresponds to Figure 4.8

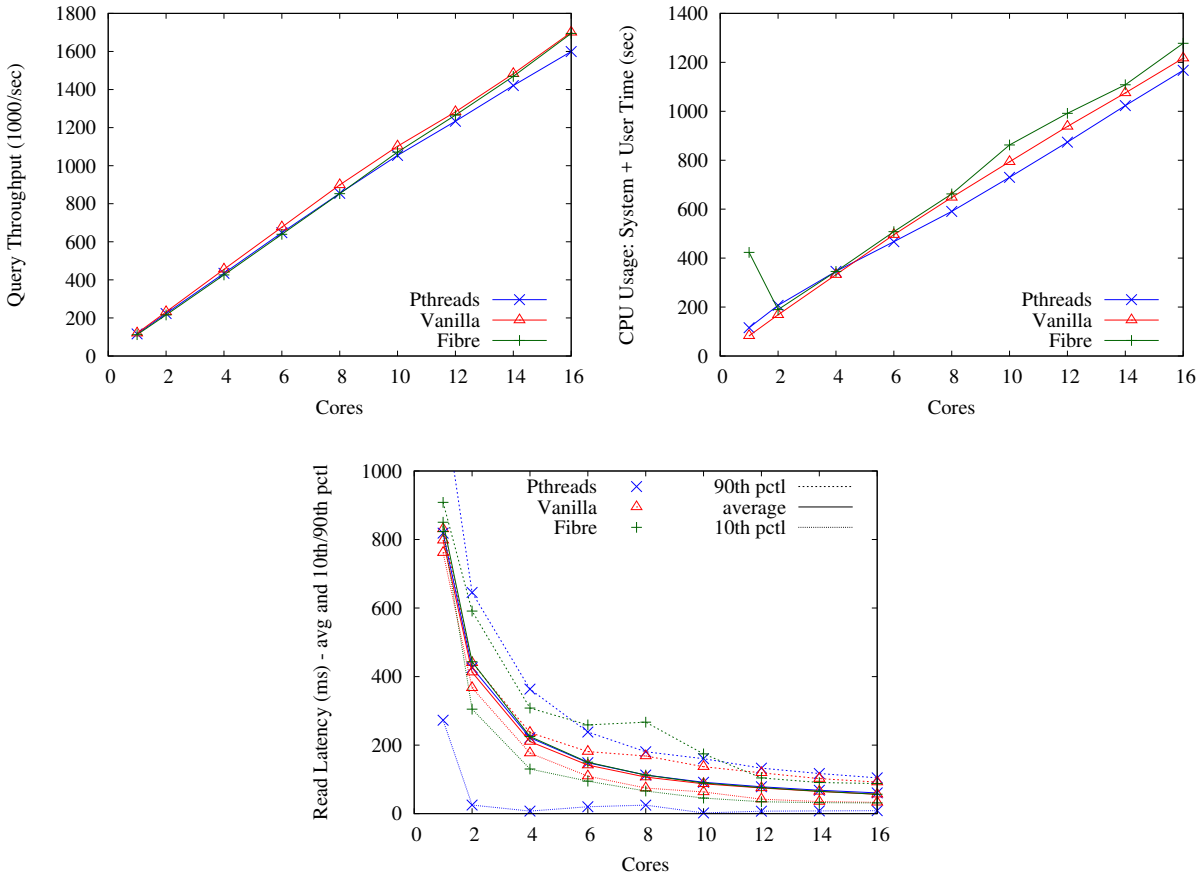


Figure B.5: Memcached - Core Scalability on Intel, Corresponds to Figure 4.10 (6,000 connections, write ratio 0.1)

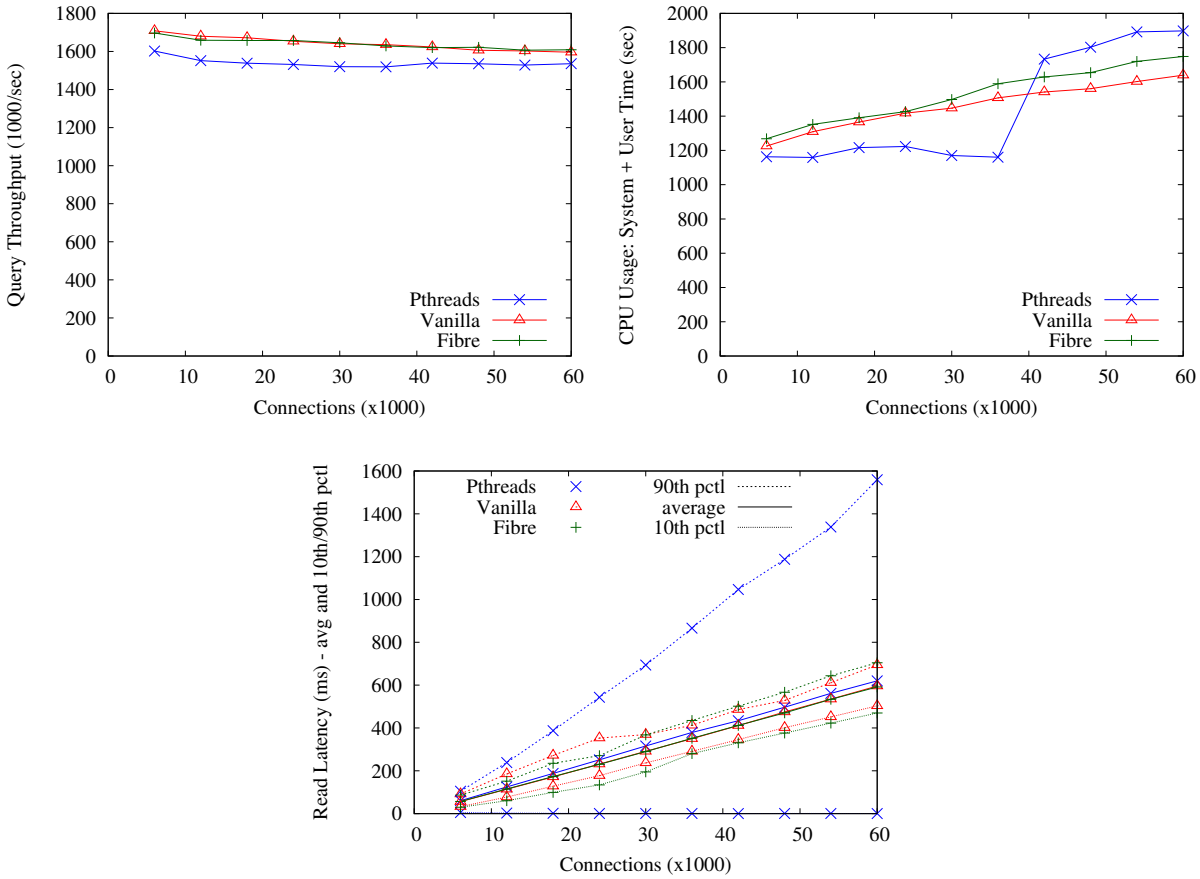


Figure B.6: Memcached - Connection Scalability on Intel, Corresponds to Figure 4.11 (16 cores, write ratio 0.1)

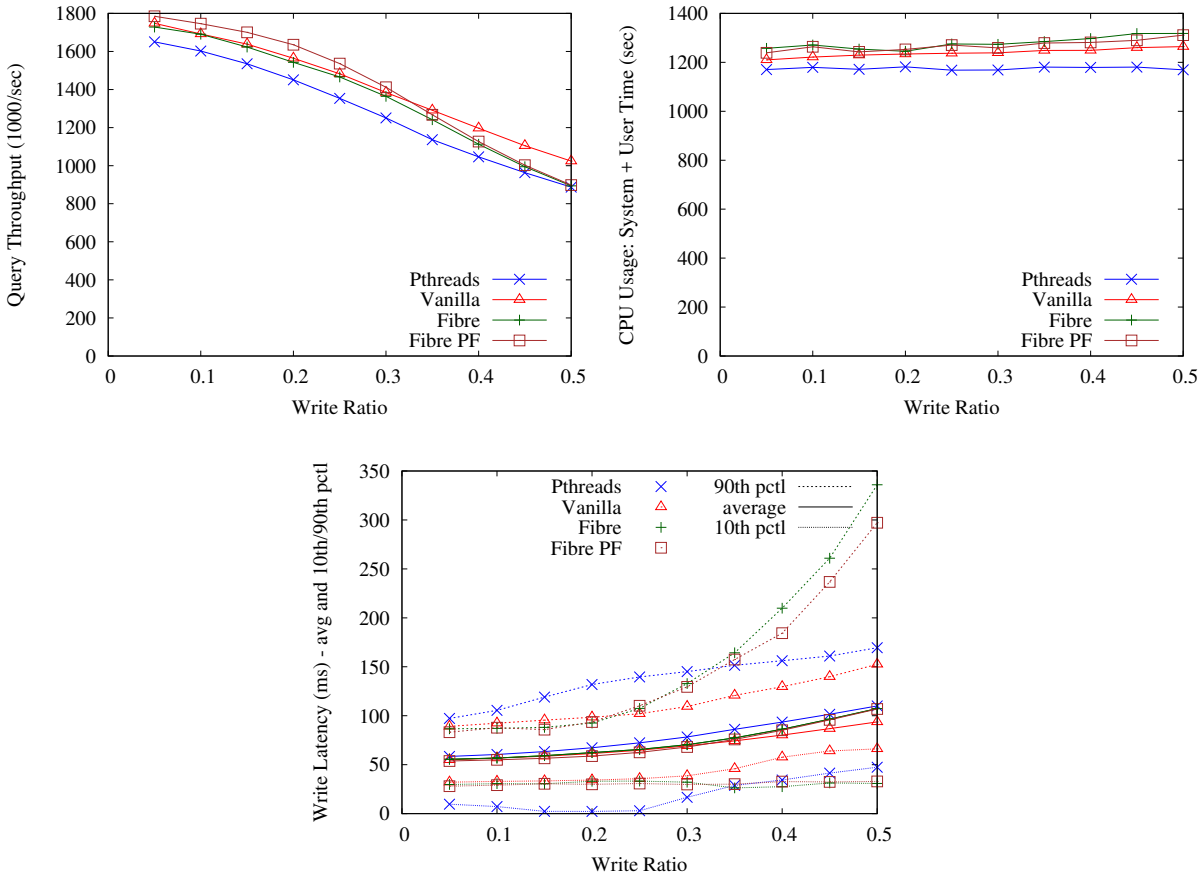


Figure B.7: Memcached - Read/Write Scalability With Blocking Mutex on Intel, Corresponds to Figure 4.12 (16 cores, 6,000 connections)

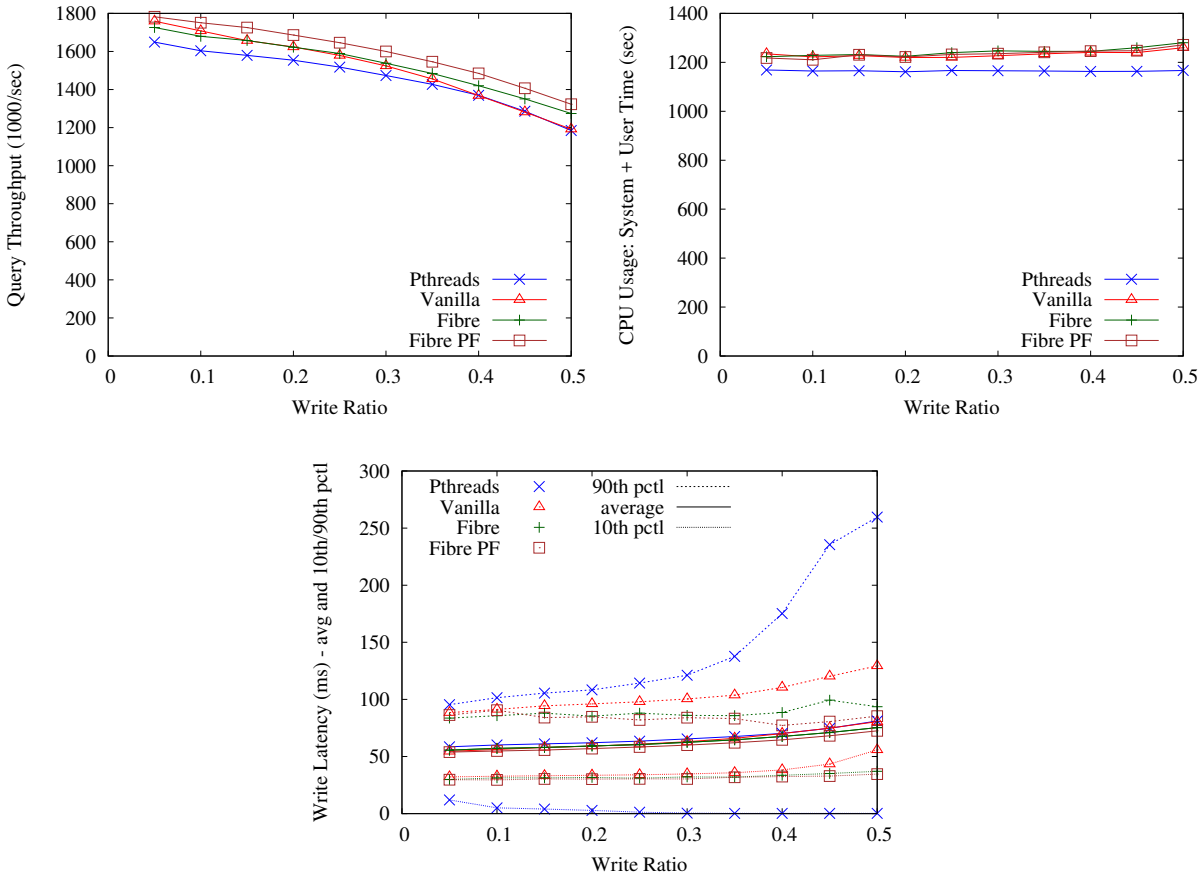


Figure B.8: Memcached - Read/Write Scalability With Spin-block Mutex on Intel, Corresponds to Figure 4.13 (16 cores, 6,000 connections)

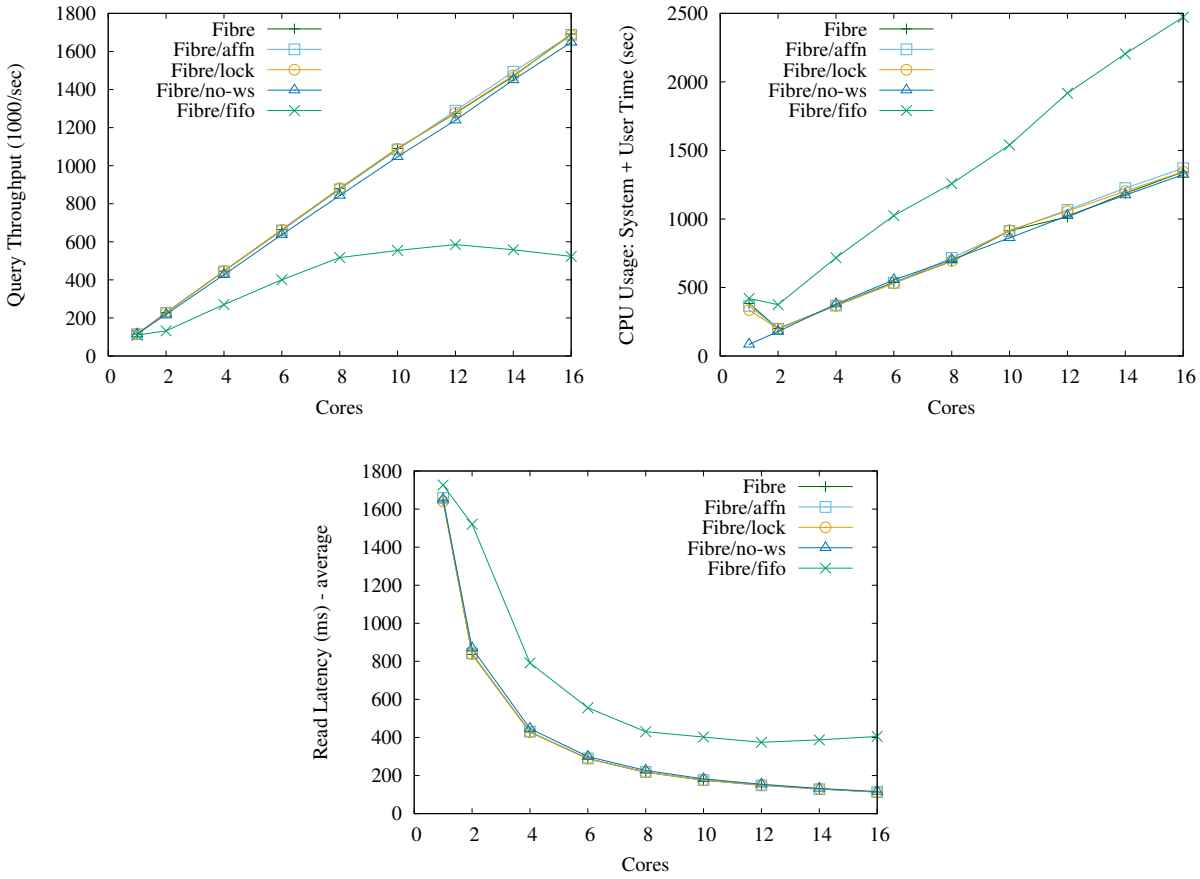


Figure B.9: Memcached - Runtime Variations on Intel, Corresponds to Figure 4.14 (12,000 connections, write ratio 0.1)

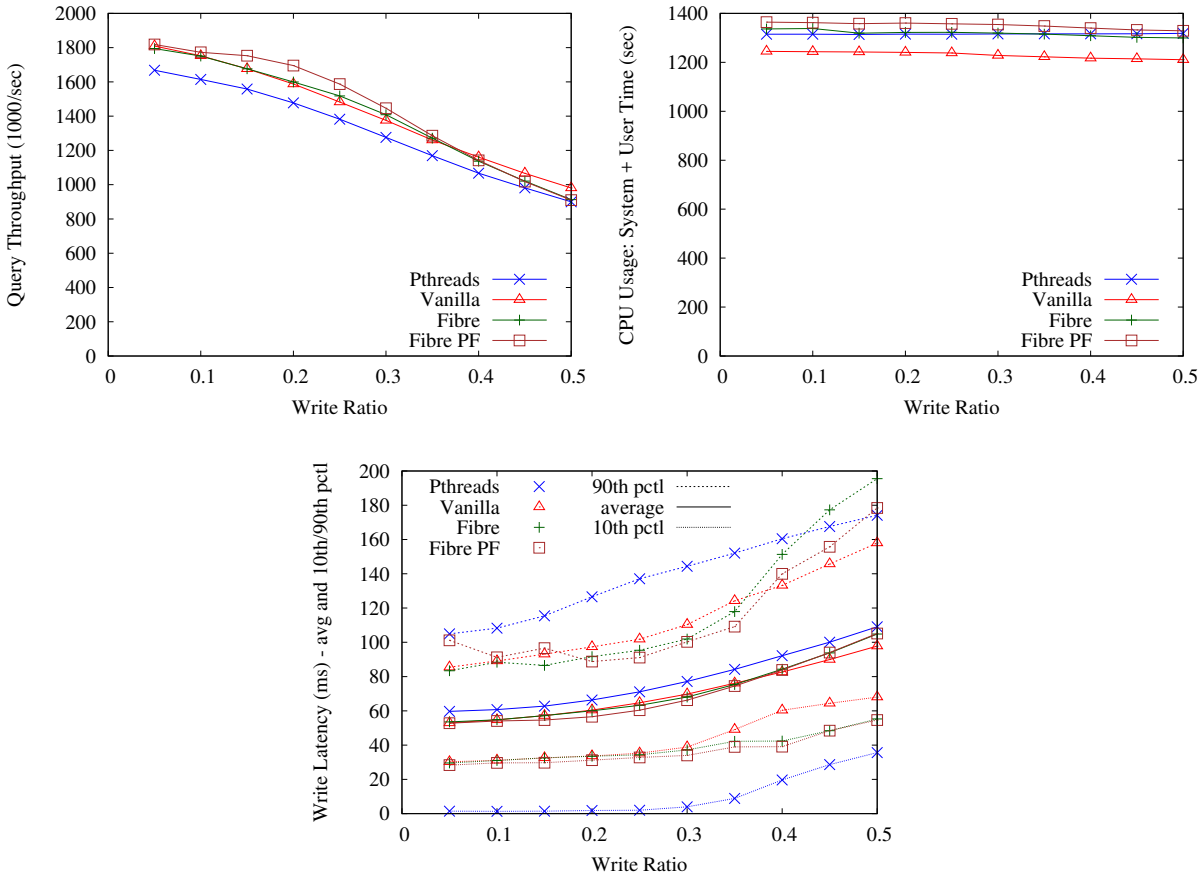


Figure B.10: Memcached - Facebook Read/Write Scalability With Blocking Mutex on Intel, Corresponds to Figure 4.15 (16 cores, 6,000 connections)

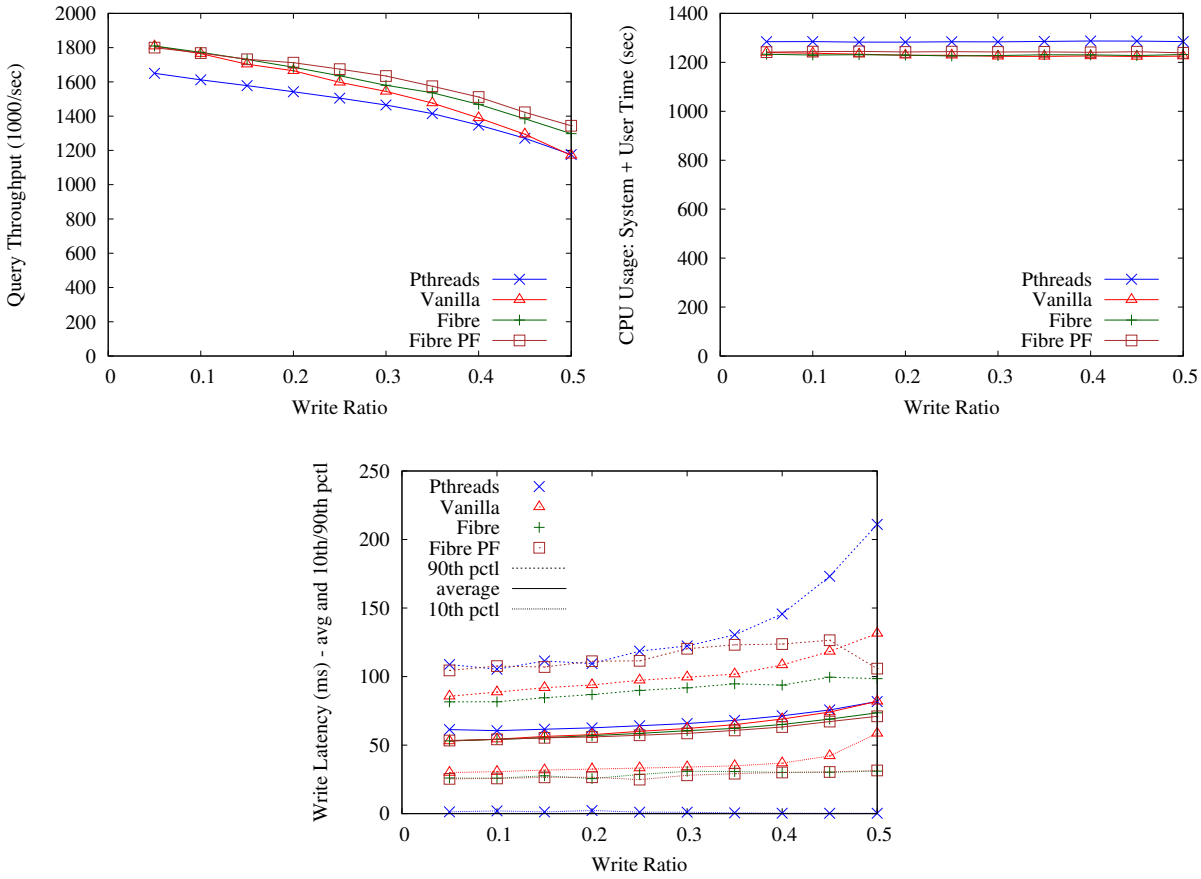


Figure B.11: Memcached - Facebook Read/Write Scalability With Spin-block Mutex on Intel, Corresponds to Figure 4.16 (16 cores, 6,000 connections)

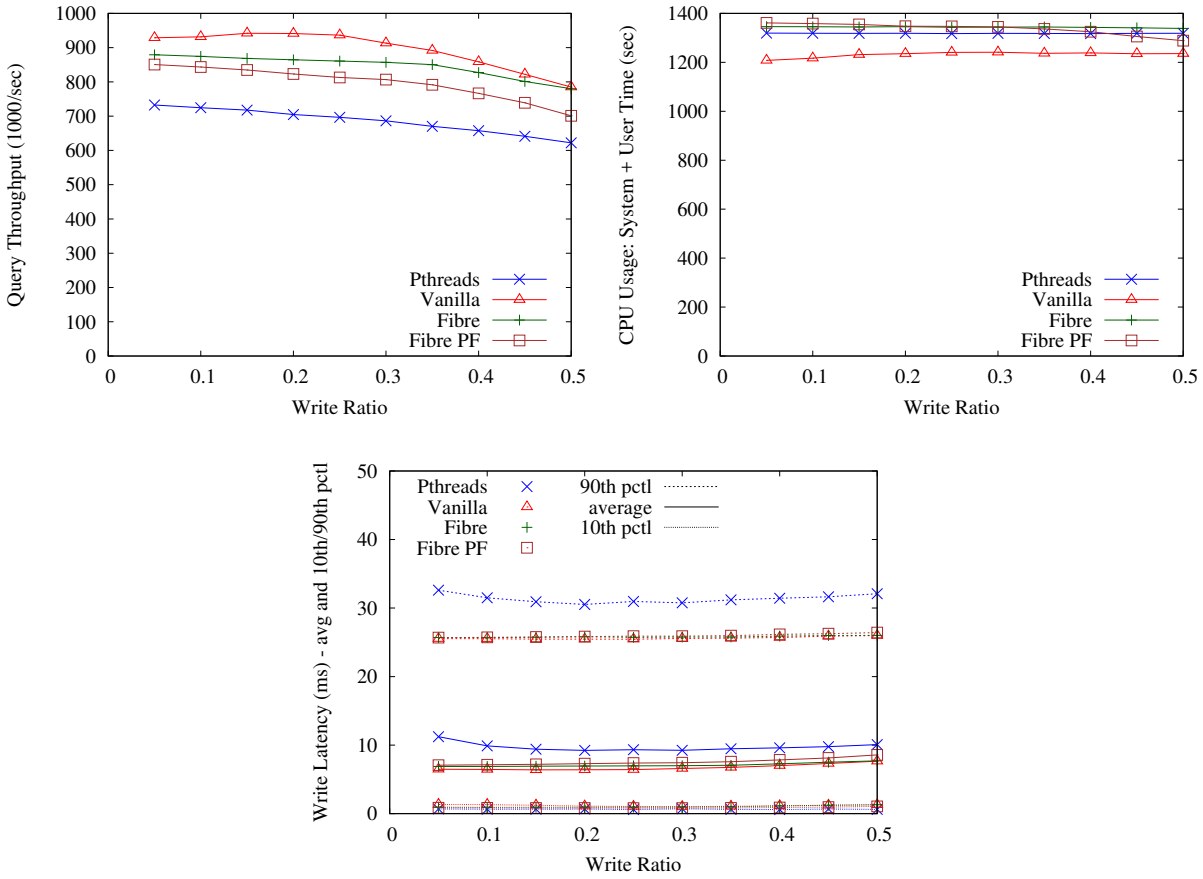


Figure B.12: Memcached - Facebook Pipeline Depth 1 With Blocking Mutex on Intel, Corresponds to Figure 4.17 (16 cores, 6,000 connections)

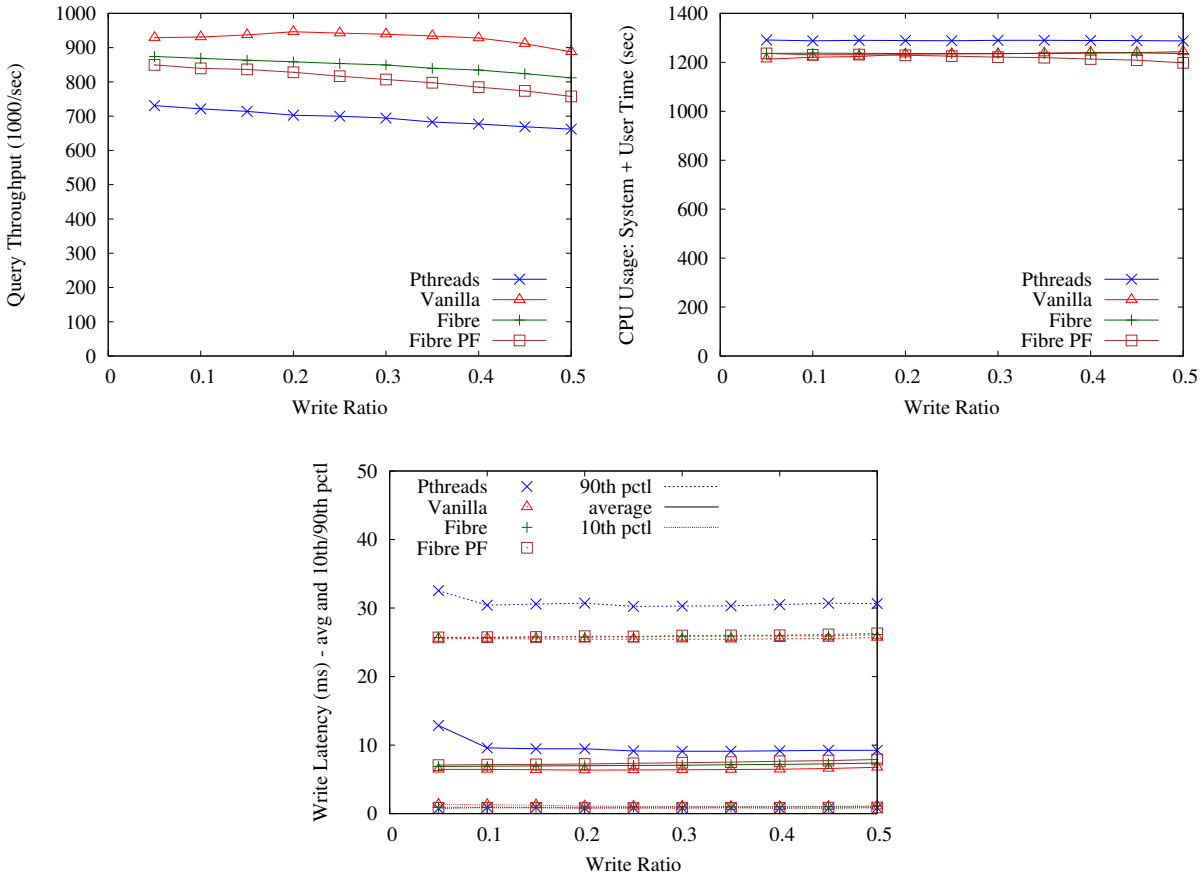


Figure B.13: Memcached - Facebook Pipeline Depth 1 With Spin-block Mutex on Intel, Corresponds to Figure 4.18 (16 cores, 6,000 connections)