The C∀ Scheduler

by

Thierry Delisle

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, 2022

© Thierry Delisle 2022

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:	Doug Lea Professor, Computer Science Department State University of New York at Oswego
Supervisor(s):	Peter Buhr Associate Professor, School of Computer Science University of Waterloo
Internal Member:	Trevor Brown Assistant Professor, School of Computer Science University of Waterloo
Internal Member:	Martin Karsten Professor, School of Computer Science University of Waterloo
Internal-External Member:	Patrick Lam Associate Professor, Department of Electrical and Computer Engineering University of Waterloo

Author's Declaration

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

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

Abstract

User-Level threading (M:N) is gaining popularity over kernel-level threading (1:1) in many programming languages. The user threading approach is often a better mechanism to express complex concurrent applications by efficiently running 10,000+ threads on multicore systems. Indeed, over-partitioning into small work-units with user threading significantly eases load balancing, while simultaneously providing advanced synchronization and mutual exclusion capabilities. To manage these high levels of concurrency, the underlying runtime must efficiently schedule many user threads across a few kernel threads; which raises the question of how many kernel threads are needed and should the number be dynamically reevaluated. Furthermore, scheduling must prevent kernel threads from blocking, otherwise user-thread parallelism drops. When user-threading parallelism does drop, how and when should idle kernel-level threads be put to sleep to avoid wasting CPU resources? Finally, the scheduling system must provide fairness to prevent a user thread from monopolizing a kernel thread; otherwise, other user threads can experience short/long term starvation or kernel threads can deadlock waiting for events to occur on busy kernel threads.

This thesis analyses multiple scheduler systems, where each system attempts to fulfill the requirements for user-level threading. The predominant technique for managing high levels of concurrency is sharding the ready queue with one queue per kernel-level thread and using some form of work stealing/sharing to dynamically rebalance workload shifts. Preventing kernel blocking is accomplished by transforming kernel locks and I/O operations into user-level operations that do not block the kernel thread or spin up new kernel threads to manage the blocking. Fairness is handled through preemption and/or ad-hoc solutions, which leads to coarse-grained fairness with some pathological cases.

After examining, selecting and testing specific approaches to these scheduling issues, a complete implementation was created and tested in the C∀ (C-for-all) runtime system. C∀ is a modern extension of C using user-level threading as its fundamental threading model. As one of its primary goals, C∀ aims to offer increased safety and productivity without sacrificing performance. The new scheduler achieves this goal by demonstrating equivalent performance to work-stealing schedulers while offering better fairness. The implementation uses several optimizations that successfully balance the cost of fairness against performance; some of these optimizations rely on interesting hardware optimizations present on modern CPUs. The new scheduler also includes support for implicit nonblocking I/O, allowing applications to have more user-threads blocking on I/O operations than there are kernel-level threads. The implementation is based on `io_uring`, a recent addition to the Linux kernel, and achieves the same performance and fairness as systems using `select`, `epoll`, *etc.* To complete the scheduler, an idle sleep mechanism is implemented that significantly reduces wasted CPU cycles, which are then available outside the application.

Acknowledgements

I would like to thank my supervisor, Professor Peter Buhr, for his guidance through my degree as well as the editing of this document.

I would like to thank Professors Martin Karsten and Trevor Brown, for reading my thesis and providing helpful feedback.

Thanks to Andrew Beach, Michael Brooks, Colby Parsons, Mubeen Zulfiqar, Fangren Yu and Jiada Liang for their work on the C∀ project as well as all the discussions which have helped me concretize the ideas in this thesis.

Finally, I acknowledge that this has been possible thanks to the financial help offered by the David R. Cheriton School of Computer Science, the corporate partnership with Huawei Ltd. and the Natural Sciences and Engineering Research Council.

Table of Contents

List of Figures x

List of Tables xii

List of Abbreviations xiii

I Introduction 1

1 Introduction 2

1.1 Scheduling 3
1.2 C∀ programming language 5
1.3 Contributions 6

2 Previous Work 7

2.1 Naming Convention 7
2.2 Static Scheduling 8
2.3 Dynamic Scheduling 8
2.3.1 Explicitly Informed Dynamic Schedulers 8
2.3.2 Uninformed and Self-Informed Dynamic Schedulers 9
2.4 Work Stealing 9
2.4.1 Theoretical Results 10
2.5 Preemption 10
2.6 Production Schedulers 11
2.6.1 Operating System Schedulers 11
2.6.2 User-Level Schedulers 12

3 C∀ Runtime 15
3.1 C Threading 15
3.2 M:N Threading 15
3.3 Clusters 16
3.4 Scheduling 16
3.5 I/O 16
3.6 Interoperating with C 17

II Design 19

4 Scheduling Core 20
4.1 Design Goals 20
4.1.1 Fairness Goals 21
4.1.2 Fairness vs Scheduler Locality 22
4.1.3 Performance Challenges 22
4.2 Inspirations 24
4.2.1 Work-Stealing 24
4.2.2 Relaxed-FIFO 24
4.3 Relaxed-FIFO++ 25
4.3.1 Dynamic Entropy 26
4.4 Work Stealing++ 26
4.4.1 Redundant Timestamps 29
4.4.2 Per CPU Sharding 30
4.4.3 Topological Work Stealing 32

5 User Level I/O 33
5.1 Kernel Interface 33
5.1.1 `O_NONBLOCK` 33
5.1.2 POSIX asynchronous I/O (AIO) 35
5.1.3 `io_uring` 35
5.1.4 Extra Kernel Threads 36
5.1.5 Discussion 36
5.2 Event-Engine 37

5.2.1 io_uring in depth 37
5.2.2 Multiplexing I/O: Submission 39
5.3 Interface 44
5.3.1 Replacement 45
5.3.2 Synchronous Extension 45
5.3.3 Asynchronous Extension 45
5.3.4 Direct io_uring Interface 46

6 Scheduling in practice **47**
6.1 Manual Resizing 47
6.1.1 Read-Copy-Update 48
6.1.2 Readers-Writer Lock 48
6.2 Idle-Sleep 49
6.3 Sleeping 50
6.3.1 pthread_mutex/pthread_cond 50
6.3.2 io_uring and Epoll 50
6.3.3 Event FDs 51
6.4 Tracking Sleepers 51
6.4.1 Sleepers List 52
6.4.2 Reducing Latency 52

III Evaluation **55**

7 Micro-Benchmarks **56**
7.1 Benchmark Environment 56
7.2 Experimental setup 57
7.3 Cycle 57
7.3.1 Results 58
7.4 Yield 61
7.4.1 Results 61
7.5 Churn 63
7.5.1 Results 65
7.6 Locality 68

7.6.1 Results 69
7.7 Transfer 72
7.7.1 Results 73

8 Macro-Benchmarks 76
8.1 Memcached 76
8.1.1 Benchmark Environment 77
8.1.2 Memcached threading 77
8.1.3 Throughput 78
8.1.4 Tail Latency 78
8.1.5 Update rate 80
8.2 Static Web-Server 80
8.2.1 NGINX threading 82
8.2.2 C∀ web server 82
8.2.3 Benchmark Environment 83
8.2.4 Throughput 84
8.2.5 Disk Operations 85

IV Conclusion & Annexes 88

9 Conclusion 89
9.1 Goals 90
9.2 Future Work 91
9.2.1 Idle Sleep 91
9.2.2 CPU Workloads 92
9.2.3 Hardware 92

References 93

Glossary 100

List of Figures

3.1 Overview of the C∀ runtime . . . 16

4.1 Fairness vs Locality graph . . . 23
4.2 Base C∀ design . . . 27
4.3 C∀ design with Moving Average . . . 28
4.4 C∀ design with Redundant Timestamps . . . 30
4.5 CPU design with wide L3 sharing . . . 31
4.6 CPU design with a narrower L3 sharing . . . 31

5.1 Overview of `io_uring` . . . 37
5.2 Partitioned ring buffer . . . 41

6.1 Specialized Readers-Writer Lock . . . 49
6.2 Basic Idle Sleep Data Structure . . . 52
6.3 Improved Idle-Sleep Data Structure . . . 53
6.4 Improved Idle-Sleep Latency . . . 54
6.5 Low-latency Idle Sleep Data Structure . . . 54

7.1 Cycle benchmark . . . 58
7.2 Cycle Benchmark: Pseudo Code . . . 59
7.3 Cycle Benchmark on Intel . . . 59
7.4 Cycle Benchmark on AMD . . . 60
7.5 Yield Benchmark: Pseudo Code . . . 62
7.6 Yield Benchmark on Intel . . . 62
7.7 Yield Benchmark on AMD . . . 64
7.8 Churn Benchmark: Pseudo Code . . . 66
7.9 Churn Benchmark on Intel . . . 66

7.10 Churn Benchmark on AMD 67
7.11 Locality Benchmark: Pseudo Code 69
7.12 Locality Benchmark on Intel 70
7.13 Locality Benchmark on AMD 71
7.14 Transfer Benchmark: Pseudo Code 72

8.1 Memcached Benchmark: Throughput 79
8.2 Memcached Benchmark: 99th Percentile Latency 79
8.3 Throughput and Latency results at different update rates (percentage of writes). . 81
8.4 Static web server Benchmark: Throughput 86

List of Tables

7.1 Transfer Benchmark on Intel and AMD 74

8.1 Cumulative memory for requests by file size 85

List of Abbreviations

API Application Programming Interface 5, 36, 39

FIFO First-In, First-Out 13, 16, 24, 101

I/O Input and Output iv, vii, viii, 5, 6, 8, 16, 17, 33–45, 47, 50, 51, 76, 80, 82, 90, 91

NUMA Non-Uniform Memory Access 10, 101

PRNG Pseudo Random Number Generator 26

Part I

Introduction

Chapter 1

Introduction

User-level threading (M:N) is gaining popularity over kernel-level threading (1:1) in many programming languages, like Go [53], Java's Project Loom [1] and Kotlin [2]. The user threading approach is often a better mechanism to express complex concurrent applications by efficiently running 10,000+ threads on multicore systems. Indeed, over-partitioning into small work units with user threading significantly eases load balancing, while simultaneously providing advanced synchronization and mutual exclusion capabilities. To manage these high levels of concurrency, the underlying runtime must efficiently schedule many user threads across a few kernel threads; which raises the question of how many kernel threads are needed and should the number be dynamically reevaluated. Furthermore, scheduling must prevent kernel threads from blocking, otherwise user-thread parallelism drops. When user-threading parallelism does drop, how and when should idle kernel-threads be put to sleep to avoid wasting CPU resources? Finally, the scheduling system must provide fairness to prevent a user thread from monopolizing a kernel thread; otherwise, other user threads can experience short/long term starvation or kernel threads can deadlock waiting for events to occur on busy kernel threads.

This thesis analyzes multiple scheduler systems, where each system attempts to fulfill the requirements for user-level threading. The predominant technique for managing high levels of concurrency is sharding the ready queue with one queue per kernel thread and using some form of work stealing/sharing to dynamically rebalance workload shifts. Preventing kernel blocking is accomplished by transforming kernel locks and I/O operations into user-level operations that do not block the kernel thread or spin up new kernel threads to manage the blocking. Fairness is handled through preemption and/or ad hoc solutions, which leads to coarse-grained fairness with some pathological cases.

After examining, testing and selecting specific approaches to these scheduling issues, a new scheduler was created and tested in the C∀ (C-for-all) user-threading runtime system. The goal of the new scheduler is to offer increased safety and productivity without sacrificing performance. The quality of the new scheduler is demonstrated by comparing it with other user-threading work-stealing schedulers with the aim of showing equivalent or better performance while offering better fairness.

Chapter 1 defines scheduling and its general goals. Chapter 2 discusses how scheduler implementations attempt to achieve these goals, but all implementations optimize some workloads better than others. Chapter 3 presents the relevant aspects of the C∀ runtime system that have a significant effect on the new scheduler design and implementation. Chapter 4 analyses

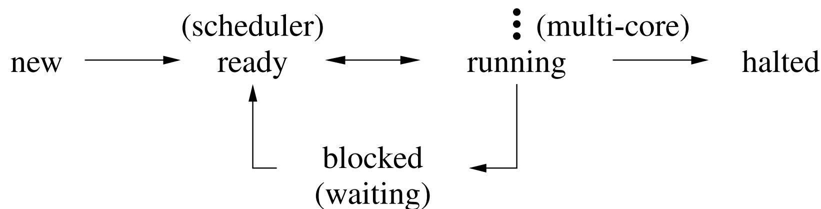
different scheduler approaches while looking for scheduler mechanisms that provide both performance and fairness. Chapter 5 covers the complex mechanisms that must be used to achieve nonblocking I/O to prevent the blocking of kernel-level threads. Chapter 6 presents the mechanisms needed to adjust the amount of parallelism, both manually and automatically. Chapters 7 and 8 present micro and macro benchmarks used to evaluate and compare the new scheduler with similar schedulers.

1.1 Scheduling

Computer systems share multiple resources across many threads of execution, even on single-user computers like laptops or smartphones. On a computer system with multiple processors and work units (routines, coroutines, threads, programs, *etc*.), there exists the problem of mapping many different kinds of work units onto many different kinds of processors efficiently, called *scheduling*. Scheduling systems are normally *open*, meaning new work arrives from an external source or is randomly spawned from an existing work unit[1].

In general, work units without threads, like routines and coroutines, are self-scheduling, while work units with threads, like tasks and programs, are scheduled. For scheduled work-units, a scheduler takes a sequence of threads and attempts to run them to completion, subject to shared resource restrictions and utilization. In an open system, a general-purpose dynamic scheduler cannot anticipate work requests, so its performance is rarely optimal. Even with complete knowledge of arrival order and work, creating an optimal solution is a bin packing problem [103]. However, optimal solutions are often not required: schedulers often produce excellent solutions, without needing optimality, by taking advantage of regularities in work patterns.

Scheduling occurs at discrete points when there are transitions in a system. For example, a thread cycles through the following transitions during its execution.



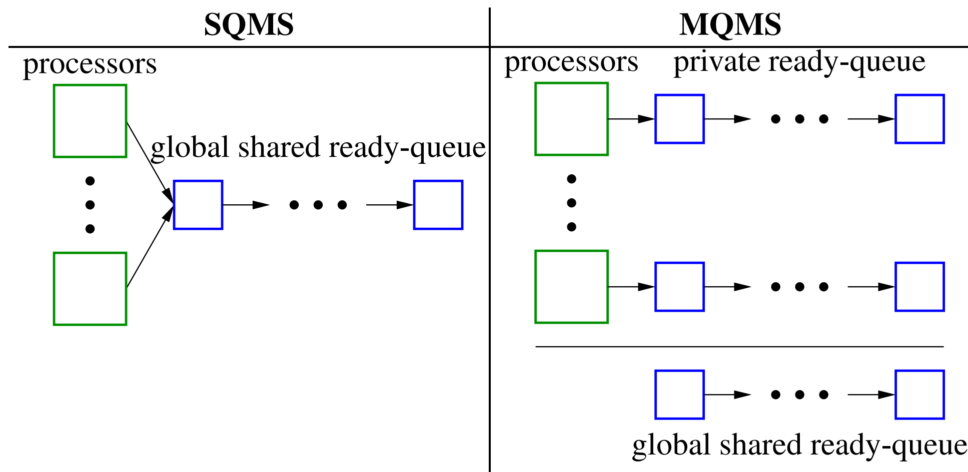
These *state transition*s are initiated in response to events, *e.g*., blocking, interrupts, errors:

- entering the system (new → ready)
- scheduler assigns a thread to a computing resource, *e.g*., CPU (ready → running)
- timer alarm for preemption (running → ready)
- long-term delay versus spinning (running → blocked)
- completion of delay, *e.g*., network or I/O completion (blocked → ready)
- normal completion or error, *e.g*., segment fault (running → halted)

[1]Open systems constrasts to *closed* systems, where work never arrives from external sources. This definition is a extension of open/closed systems in the field of thermodynamics.

Key to scheduling is that a thread cannot bypass the "ready" state during a transition so the scheduler maintains complete control of the system, *i.e.*, no self-scheduling among threads.

When the workload exceeds the capacity of the processors, *i.e.*, work cannot be executed immediately, it is placed on a queue for subsequent service, called a *ready queue*. Ready queues organize threads for scheduling, which indirectly organizes the work to be performed. The structure of ready queues can take many different forms, where the basic two are the single-queue multi-server (SQMS) and the multi-queue multi-server (MQMS).



Beyond these two schedulers are a host of options, *e.g.*, adding a global shared queue to MQMS or adding multiple private queues with distinct characteristics.

Once there are multiple resources and ready queues, a scheduler is faced with three major optimization criteria:

1. *load balancing*: available work is distributed so no processor is idle when work is available.
   Eventual progress for each work unit is often an important consideration, *i.e.*, no starvation.
2. *affinity*: processors access state through a complex memory hierarchy, so it is advantageous to keep a work unit's state on a single or closely bound set of processors.
   Essentially, all multiprocessor computers have non-uniform memory access (NUMA), with one or more quantized steps to access data at different levels in the memory hierarchy. When a system has a large number of independently executing threads, affinity becomes difficult because of *thread churn*. That is, threads must be scheduled on different processors to obtain high processor utilization because the number of threads $\ggg$ processors.
3. *contention*: safe access of shared objects by multiple processors requires mutual exclusion in some form, generally locking.[2] Mutual exclusion cost and latency increase significantly with the number of processors accessing a shared object.

At a high-level, scheduling is considered zero-sum game as computer processors normally have a fixed, maximum number of cycles per unit time.[3] This almost invariably leads to schedulers needing to pick some threads over others, opening the door to fairness problems. However,

[2] Lock-free data-structures do not involve locking but incur similar costs to achieve mutual exclusion.

[3] Frequency scaling and turbo-boost add a degree of complexity that can be ignored in this discussion without loss of generality.

at a lower-level, schedulers can make inefficient or incorrect decisions leading to strictly worse outcomes than what the theoretical zero-sum game suggests. Since it can be difficult to avoid these poor decisions, schedulers are generally a series of compromises, occasionally with some static or dynamic tuning parameters to enhance specific workload patterns. For example, SQMS has perfect load-balancing but poor affinity and high contention by the processors, because of the single queue. While MQMS has poor load-balancing but perfect affinity and no contention, because each processor has its own queue.

Significant research effort has looked at load balancing by stealing/sharing work units among queues: when a ready queue is too short or long, respectively, load stealing/sharing schedulers attempt to push/pull work units to/from other ready queues. These approaches attempt to perform better load-balancing at the cost of affinity and contention. However, *all* approaches come at a cost, but not all compromises are necessarily equivalent, especially across workloads. Hence, some schedulers perform very well for specific workloads, while others offer acceptable performance over a wider range of workloads.

1.2 C∀ programming language

The C∀ programming language [26, 61] extends the C programming language by adding modern safety and productivity features, while maintaining backwards compatibility. Among its productivity features, C∀ supports user-level threading [32] as its fundamental threading model allowing programmers to easily write modern concurrent and parallel programs. My previous master's thesis on concurrency in C∀ focused on features and interfaces [31]. This Ph.D. thesis focuses on performance, introducing API changes only when required by performance considerations. Specifically, this work concentrates on advanced thread and I/O scheduling. Prior to this work, the C∀ runtime used a strict SQMS ready-queue and provided no nonblocking I/O capabilities at the user-thread level.[4]

Since C∀ attempts to improve the safety and productivity of C, the new scheduler presented in this thesis attempts to achieve the same goals. More specifically, safety and productivity for scheduling mean supporting a wide range of workloads, so that programmers can rely on progress guarantees (safety) and more easily achieve acceptable performance (productivity). The new scheduler also includes support for implicit nonblocking I/O, allowing applications to have more user-threads blocking on I/O operations than there are kernel-level threads. To complete the scheduler, an idle sleep mechanism is implemented that significantly reduces wasted CPU cycles, which are then available outside the application.

As a research project, this work runs exclusively on newer versions of the Linux operating system and gcc/clang compilers. The new scheduler implementation uses several optimizations to successfully balance the cost of fairness against performance; some of these optimizations rely on interesting hardware optimizations only present on modern CPUs. The I/O implementation is based on the `io_uring` kernel interface, a recent addition to the Linux kernel, because it purports to handle nonblocking *file* and network I/O. This decision allowed an interesting performance and fairness comparison with other threading systems using `select`, `epoll`, *etc*. While the current

[4] C/C++ only support I/O capabilities at the kernel level, which means many I/O operations block kernel-level threads, reducing parallelism at the user level.

C∀ release supports older versions of Linux ($\geq$ Ubuntu 16.04) and gcc/clang compilers ($\geq$ gcc 6.0), it is not the purpose of this project to find workarounds in these older systems to provide backwards compatibility. The hope is that these new features will soon become mainstream features.

1.3 Contributions

This work provides the following scheduling contributions for advanced user-level threading runtime systems:

1. A scalable scheduling algorithm that offers progress guarantees.
2. Support for user-level I/O capabilities based on Linux's `io_uring`.
3. An algorithm for load-balancing and idle sleep of processors, including NUMA awareness.
4. A mechanism for adding fairness on top of the MQMS algorithm through helping, used both for scalable scheduling algorithm and the user-level I/O.
5. An optimization of the helping mechanism for load balancing to reduce scheduling costs.
6. An optimization for the alternative relaxed-list for load balancing to reduce scheduling costs in embarrassingly parallel cases.

Chapter 2

Previous Work

As stated, scheduling is the process of assigning resources to incoming requests, where the common example is assigning available workers to work requests or vice versa. Common scheduling examples in Computer Science are: operating systems and hypervisors schedule available CPUs, NICs schedule available bandwidth, virtual memory and memory allocator schedule available storage, *etc*. Scheduling is also common in most other fields; *e.g.*, in assembly lines, assigning parts to line workers is a form of scheduling.

In general, *selecting* a scheduling algorithm depends on how much information is available to the scheduler. Workloads that are well known, consistent, and homogeneous can benefit from a scheduler that is optimized to use this information, while ill-defined, inconsistent, heterogeneous workloads require general non-optimal algorithms. A secondary aspect is how much information can be gathered versus how much information must be given as part of the scheduler input. This information adds to the spectrum of scheduling algorithms, going from static schedulers that are well informed from the start, to schedulers that gather most of the information needed, to schedulers that can only rely on very limited information. This description includes both information about each request, *e.g.*, time to complete or resources needed, and information about the relationships among requests, *e.g.*, whether some requests must be completed before another request starts.

Scheduling physical resources, *e.g.*, in an assembly line, is generally amenable to using well-informed scheduling since information can be gathered much faster than the physical resources can be assigned, and workloads are likely to stay stable for long periods. When a faster pace is needed and changes are much more frequent, then gathering information on workloads, up-front or live, can become much more limiting and more general schedulers are needed.

2.1 Naming Convention

Scheduling has been studied by various communities, concentrating on different incarnations of the same problems. As a result, there are no standard naming conventions for scheduling that are respected across these communities. This document uses the term *Thread* to refer to the abstract objects being scheduled and the term *Processor* to refer to the concrete objects executing these threads.

2.2 Static Scheduling

Static schedulers require thread dependencies and costs to be explicitly and exhaustively specified prior to scheduling. The scheduler then processes this input ahead of time and produces a *schedule* the system follows during execution. This approach is popular in real-time systems since the need for strong guarantees justifies the cost of determining and supplying this information. In general, static schedulers are less relevant to this project because they require input from the programmers that the C∀ programming language does not have as part of its concurrency semantics. Specifying this information explicitly adds a significant burden to the programmer and reduces flexibility. For this reason, the C∀ scheduler does not require this information.

2.3 Dynamic Scheduling

Dynamic schedulers detect thread dependencies and costs during scheduling, if at all. This detection takes the form of observing new threads in the system and determining dependencies from their behaviour, where a thread suspends or halts dynamically when it detects unfulfilled dependencies. Furthermore, each thread has the responsibility of adding dependent threads back into the system once dependencies are fulfilled. As a consequence, the scheduler often has an incomplete view of the system, seeing only threads with no pending dependencies.

2.3.1 Explicitly Informed Dynamic Schedulers

While dynamic schedulers may not have an exhaustive list of dependencies for a thread, some information may be available about each thread, *e.g.*, expected duration, required resources, relative importance, *etc*. When available, a scheduler can then use this information to direct scheduling decisions. For example, when scheduling in a cloud computing context, threads will commonly have extra information that was manually entered, *e.g.*, caps on compute time or I/O usage. However, in the context of user-level threading, most programmers do not determine or even *predict* this information; at best, the scheduler has only some imprecise information provided by the programmer, *e.g.*, indicating a thread takes approximately 3–7 seconds to complete, rather than exactly 5 seconds. Providing this kind of information is a significant programmer burden, especially if the information does not scale with the number of threads and their complexity. For example, providing an exhaustive list of files read by 5 threads is an easier requirement than providing an exhaustive list of memory addresses accessed by 10,000 independent threads.

Since the goal of this thesis is to provide a scheduler as a replacement for C∀'s existing *uninformed* scheduler, explicitly informed schedulers are less relevant to this project. Nevertheless, some strategies are worth mentioning.

Priority Scheduling

A common approach to direct the scheduling algorithm is to add information about thread priority. Each thread is given a priority, and higher-priority threads are preferred to lower-priority

ones. The simplest priority scheduling algorithm is to require that every thread have a distinct pre-established priority and always run the available threads with the highest priority. Asking programmers to provide an exhaustive set of unique priorities can be prohibitive when the system has a large number of threads. It can therefore be desirable for schedulers to support threads with identical priorities and/or automatically set and adjust priorities for threads. Most common operating systems use some variant on priorities with overlaps and dynamic priority adjustments. For example, Microsoft Windows uses a pair of priorities [84], one specified by users out of ten possible options and one adjusted by the system.

2.3.2 Uninformed and Self-Informed Dynamic Schedulers

Several scheduling algorithms do not require programmers to provide additional information on each thread, and instead, make scheduling decisions based solely on internal state and/or information implicitly gathered by the scheduler.

Feedback Scheduling

As mentioned, schedulers may also gather information about each thread to direct their decisions. This design effectively moves the scheduler into the realm of *Control Theory* [100]. This information gathering does not generally involve programmers, and as such, does not increase programmer burden the same way explicitly provided information may. However, some feedback schedulers do allow programmers to offer additional information on certain threads, to direct scheduling decisions. The important distinction is whether the scheduler can function without this additional information.

2.4 Work Stealing

One of the most popular scheduling algorithms in practice (see 2.6) is work stealing. This idea, introduced by [25], effectively has each worker process its local threads first but allows the possibility for other workers to steal local threads if they run out of threads. [23] introduced the more familiar incarnation of this, where each worker has a queue of threads and workers without threads steal threads from random workers[1]. Blumofe and Leiserson also prove worst-case space and time requirements for well-structured computations.

Many variations of this algorithm have been proposed over the years [111], both optimizations of existing implementations and approaches that account for new metrics.

Granularity A significant portion of early work-stealing research concentrated on *Implicit Parallelism* [98]. Since the system is responsible for splitting the work, granularity is a challenge that cannot be left to programmers, as opposed to *Explicit Parallelism*[95] where the burden can be left to programmers. In general, fine granularity is better for load balancing and coarse granularity reduces communication overhead. The best performance generally means finding a middle ground between the two. Several methods can be employed, but I believe these are less relevant for threads, which are generally explicit and more coarse-grained.

[1] The Burton and Sleep algorithm has trees of threads and steals only among neighbours.

Task Placement Another aspect of work stealing that has been studied extensively is the mapping between thread and processor. In its simplest form, work stealing assumes that all processors are interchangeable and therefore the mapping between thread and processor is not interesting. However, in real-life architectures, there are contexts where different processors can have different characteristics, which makes some mappings more interesting than others. A common example where this is statically true is architectures with NUMA. In these cases, it can be relevant to change the scheduler to be cognizant of the topology [91, 58]. Another example is energy usage, where the scheduler is modified to optimize for energy efficiency in addition/instead of performance [73, 89].

Complex Machine Architecture Another aspect that has been examined is how applicable work stealing is to different machine architectures. This is arguably strongly related to Task Placement but extends into more heterogeneous architectures. As C∀ offers no particular support for heterogeneous architectures, this is also an area that is not examined in this thesis. However, support for concurrency across heterogeneous architectures is interesting avenue for future work, at which point the literature on this topic and how it relates to scheduling will become relevant.

2.4.1 Theoretical Results

There is also a large body of research on the theoretical aspects of work stealing. These evaluate, for example, the cost of migration [81, 33], how affinity affects performance [80, 8, 82] and theoretical models for heterogeneous systems [59, 20, 36]. Blelloch et al. [22] examines the space bounds of work stealing and [21] shows that for under-loaded systems, the scheduler completes its computations in finite time, *i.e.*, is *stable*. Others show that work stealing applies to various scheduling contexts [16, 85, 86, 11, 10]. [28] also studied how randomized work-stealing affects false sharing among threads.

However, as [111] highlights, it is worth mentioning that this theoretical research has mainly focused on "fully strict" computations, *i.e.*, workloads that can be fully represented with a direct acyclic graph. It is unclear how well these distributions represent workloads in real-world scenarios.

2.5 Preemption

One last aspect of scheduling is preemption since many schedulers rely on it for some of their guarantees. Preemption is the idea of interrupting threads that have been running too long, effectively injecting suspend points into the application. There are multiple techniques to achieve this effect, but they all aim to guarantee that the suspend points in a thread are never further apart than some fixed duration. This helps schedulers guarantee that no threads unfairly monopolize a worker. Preemption can effectively be added to any scheduler. Therefore, the only interesting aspect of preemption for the design of scheduling is whether to require it.

2.6 Production Schedulers

This section presents a quick overview of several current schedulers. While these schedulers do not necessarily represent the most recent advances in scheduling, they are what is generally accessible to programmers. As such, I believe these schedulers are as relevant as those presented in published work. Both schedulers that operate in kernel space and user space are considered, as both can offer relevant insight for this project. However, real-time schedulers aim to guarantee bounded compute time in order to meet deadlines. These deadlines lead to constraints much stricter than the starvation freedom that is needed for this project. As such real-time schedulers are not considered for this work.

2.6.1 Operating System Schedulers

Operating System Schedulers tend to be fairly complex, as they generally support some amount of real time, aim to balance interactive and non-interactive threads and support multiple users sharing hardware without requiring these users to cooperate. Here are more details on a few schedulers used in the common operating systems: Linux, FreeBSD, Microsoft Windows and Apple's OS X. The information is less complete for closed source operating systems.

Linux's CFS The default scheduler used by Linux, the Completely Fair Scheduler [4, 47], is a feedback scheduler based on CPU time. For each processor, it constructs a Red-Black tree of threads waiting to run, ordering them by the amount of CPU time used. The thread that has used the least CPU time is scheduled. It also supports the concept of *Nice values*, which are effectively multiplicative factors on the CPU time used. The ordering of threads is also affected by a group-based notion of fairness, where threads belonging to groups having used less CPU time are preferred to threads belonging to groups having used more CPU time. Linux achieves load-balancing by regularly monitoring the system state [5] and using some heuristic on the load, currently CPU time used in the last millisecond plus a decayed version of the previous time slots [29].

[57] shows that Linux's CFS also does work stealing to balance the workload of each processor, but the paper argues this aspect can be improved significantly. The issues highlighted stem from Linux's need to support fairness across threads *and* across users[2], increasing the complexity.

Linux also offers a FIFO scheduler, a real-time scheduler, which runs the highest-priority threads, and a round-robin scheduler, which is an extension of the FIFO scheduler that adds fixed time slices. [6]

FreeBSD The ULE scheduler used in FreeBSD[74] is a feedback scheduler similar to Linux's CFS. It uses different data structures and heuristics but also schedules according to some combination of CPU time used and niceness values. It also periodically balances the load of the system (according to a different heuristic) but uses a simpler work stealing approach.

[2]Enforcing fairness across users means that given two users, one with a single thread and the other with one thousand threads, the user with a single thread does not receive one-thousandth of the CPU time.

Windows (OS) Microsoft's Operating System's Scheduler [27] is a feedback scheduler with priorities. It supports 32 levels of priorities, some of which are reserved for real-time and privileged applications. It schedules threads based on the highest priorities (lowest number) and how much CPU time each thread has used. The scheduler may also temporarily adjust priorities after certain effects like the completion of I/O requests.

The scheduling policy is discussed more in-depth in [75], Chapter 1 section 2.3 "Processes, Threads, and Jobs". Multicore scheduling is based on a combination of priorities and processor preference. Each thread is assigned an initial processor using a round-robin policy, called the thread's *ideal* processor. Threads are distributed among the processors according to their priority, preferring to match threads to their ideal processor and then to the last processor they ran on. This approach is a variation of work stealing, where the stealing processor restores the thread to its original processor after running it, but mixed with priorities.

Apple OS X Apple programming documentation describes its kernel scheduler as follows:

> OS X is based on Mach and BSD. [...]. It contains an advanced scheduler based on the CMU Mach 3 scheduler. [...] Mach scheduling is based on a system of run queues at various priorities.
>
> – Kernel Programming Guide [15]

There is very little documentation on the internals of this scheduler. However, the documentation does describe a feature set that is very similar to the Windows and Linux OS schedulers. Presumably, this means that the internals are also fairly similar overall.

2.6.2 User-Level Schedulers

By comparison, user-level schedulers tend to be simpler, gather fewer metrics and avoid complex notions of fairness. Part of the simplicity is due to the fact that all threads have the same user, and therefore cooperation is both feasible and probable.

Go Go's scheduler uses a randomized work-stealing algorithm that has a global run-queue (*GRQ*) and each processor (*P*) has both a fixed-size run-queue (*LRQ*) and a high-priority next "chair" holding a single element [53, 93]. Preemption is present, but only at safe points [54], which are detection code inserted at various frequent access boundaries.

The algorithm is as follows :

1. Once out of 61 times, pick 1 element from the *GRQ*.
2. Otherwise, if there is an item in the "chair" pick it.
3. Else pick an item from the *LRQ*.
    - If it is empty steal (len(*GRQ*) / #of*P*) + 1 items (max 256) from the *GRQ*
    - and steal *half* the *LRQ* of another *P* chosen randomly.

Chapter 7 uses Go as one of its comparison point in this thesis's performance evaluation.

Erlang Erlang is a functional language that supports concurrency in the form of processes: threads that share no data. It uses a kind of round-robin scheduler, with a mix of work sharing and stealing to achieve load balancing [51], where under-loaded workers steal from other workers, but overloaded workers also push work to other workers. This migration logic is directed by monitoring logic that evaluates the load a few times per second.

Intel® Threading Building Blocks *Thread Building Blocks* (TBB) is Intel's task parallelism [96] framework. It runs *jobs*, which are uninterruptible threads that must always run to completion, on a pool of worker threads. TBB's scheduler is a variation of randomized work-stealing that also supports higher-priority graph-like dependencies [45]. It schedules threads as follows (where t is the last thread completed):

> 1. The task returned by t`.execute()`
> 2. The successor of t if t was its last completed predecessor.
> 3. A task popped from the end of the thread's own queue.
> 4. A task with an affinity for the thread.
> 5. A task popped from approximately the beginning of the shared queue.
> 6. A task popped from the beginning of another randomly chosen thread's queue.
>
> – Intel® TBB documentation [45]

Quasar/Project Loom Java has two projects, Quasar [68] and Project Loom [1][3], that are attempting to introduce lightweight threading in the form of Fibers. Both projects seem to be based on the `ForkJoinPool` in Java, which appears to be a simple incarnation of randomized work-stealing [3].

Grand Central Dispatch An Apple [44] API that offers task parallelism [96]. Its distinctive aspect is multiple "Dispatch Queues", some of which are created by programmers. Each queue has its own local ordering guarantees, *e.g.*, threads on queue A are executed in *FIFO* order.

While the documentation only gives limited insight into the scheduling and load balancing approach, [43] suggests a fairly classic approach. Each processor has a queue of threads to run, called *blocks*, which are drained in FIFO. GCD also has secondary queues, called *Dispatch Queues*, with clear ordering, where executing a block ends up scheduling more blocks. In terms of semantics, these Dispatch Queues seem to be very similar to Intel® TBB `execute()` and predecessor semantics.

The similarity of API and semantics between GCD and Intel® TBB suggest the underlying scheduling algorithms are similar.

[3] It is unclear if these are distinct projects.

LibFibre LibFibre [49] is a lightweight user-level threading framework developed at the University of Waterloo. It shares a very strong resemblance to Go: using a variation of work stealing with a global queue that has a higher priority than stealing. Unlike Go, it does not have the high-priority next "chair" and its work-stealing is not randomized.

Chapter 7 uses LibFibre as one of its comparison point in this thesis's performance evaluation.

Chapter 3

C∀ Runtime

This chapter presents an overview of the capabilities of the C∀ runtime prior to this thesis work.

3.1 C Threading

C11 introduced threading features, such as the **`_Thread_local`** storage class, and libraries `stdatomic.h` and `threads.h`. Interestingly, almost a decade after the C11 standard, the most recent versions of gcc, clang, and msvc do not support the C11 include `threads.h`, indicating no interest in the C11 concurrency approach (possibly because of the recent effort to add concurrency to C++). While the C11 standard does not state a threading model, the historical association with pthreads suggests implementations would adopt kernel-level threading (1:1) [87], as C++ does. This model uses kernel-level threads to achieve parallelism and concurrency. In this model, every thread of computation maps to an object in the kernel. The kernel then has the responsibility of managing these threads, *e.g.*, creating, scheduling, blocking. A consequence of this approach is that the kernel has a perfect view of every thread executing in the system[1].

3.2 M:N Threading

Threading in C∀ is based on User-level threading, where threads are the representation of a unit of work. As such, C∀ programmers should expect these units to be fairly inexpensive, *i.e.*, programmers should be able to create a large number of threads and switch among threads liberally without many performance concerns.

The C∀ M:N threading model is implemented using many user-level threads mapped onto fewer kernel-level threads. The user-level threads have the same semantic meaning as a kernel-level threads in the 1:1 model: they represent an independent thread of execution with its own stack. The difference is that user-level threads do not have a corresponding object in the kernel; they are handled by the runtime in user space and scheduled onto kernel-level threads, referred to as processors in this document. Processors run a thread until it context switches out, it then chooses a different thread to run.

[1] This is not completely true due to primitives like `futex`es, which have a significant portion of their logic in user space.

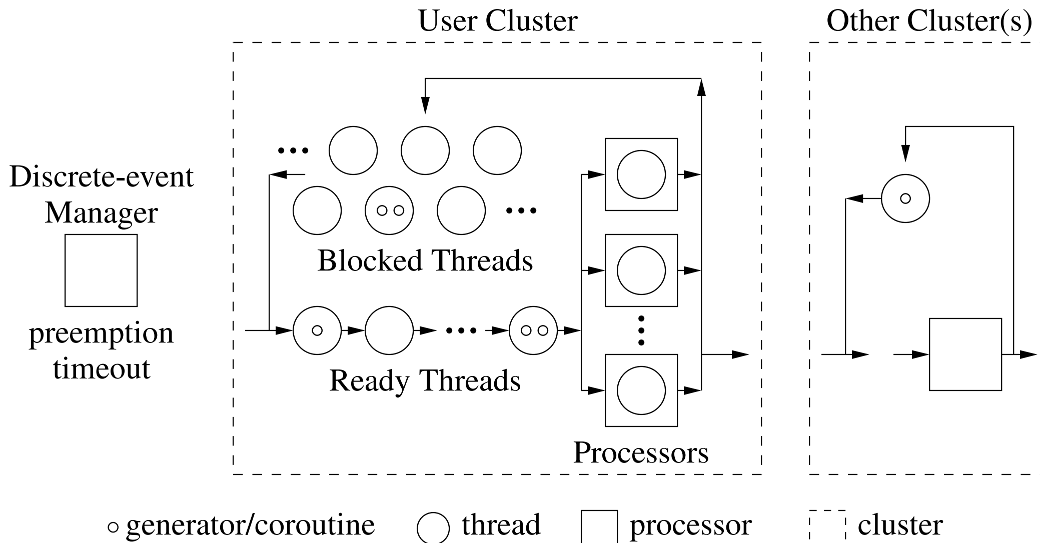


Figure 3.1: Overview of the C∀ runtime
Threads are scheduled inside a particular cluster and run on the processors that belong to the cluster. The discrete-event manager, which handles preemption and timeout, is a processor that lives outside any cluster and does not run threads.

3.3 Clusters

C∀ allows the option to group user-level threading, in the form of clusters. Both threads and processors belong to a specific cluster. Threads are only scheduled onto processors in the same cluster and scheduling is done independently of other clusters. Figure 3.1 shows an overview of the C∀ runtime, which allows programmers to tightly control parallelism. It also opens the door to handling effects like NUMA, by pinning clusters to a specific NUMA node[2].

3.4 Scheduling

The C∀ runtime previously used a FIFO ready-queue with a single lock. This setup offers perfect fairness in terms of opportunities to run. However, it offers poor scalability, since the performance of the ready queue can never be improved by adding more hardware threads and contention can cause significant performance degradation.

3.5 I/O

Prior to this work, the C∀ runtime did not have any particular support for I/O operations. While all I/O operations available in C are available in C∀, I/O operations are designed for the POSIX threading model [41]. Using these 1:1 threading operations in an M:N threading model

[2] This capability is not currently implemented in C∀, but the only hurdle left is creating a generic interface for CPU masks.

means I/O operations block processors instead of threads. While this can work in certain cases, it limits the number of concurrent operations to the number of processors rather than threads. It also means deadlock can occur because all processors are blocked even if at least one thread is ready to run. A simple example of this type of deadlock would be as follows:

> Given a simple network program with 2 threads and a single processor, one thread sends network requests to a server and the other thread waits for a response from the server. If the second thread races ahead, it may wait for responses to requests that have not been sent yet. In theory, this should not be a problem, even if the second thread waits, because the first thread is still ready to run and should be able to get CPU time to send the request. With M:N threading, while the first thread is ready, the lone processor *cannot* run the first thread if it is blocked in the I/O operation of the second thread. If this happens, the system is in a synchronization deadlock[3].

Therefore, one of the objectives of this work is to introduce *User-Level I/O*, which, like User-Level *Threading*, blocks threads rather than processors when doing I/O operations. This feature entails multiplexing the I/O operations of many threads onto fewer processors. The multiplexing requires a single processor to execute multiple I/O operations in parallel. This requirement cannot be done with operations that block processors, *i.e.*, kernel-level threads, since the first operation would prevent starting new operations for its blocking duration. Executing I/O operations in parallel requires *asynchronous* I/O, sometimes referred to as *non-blocking*, since the kernel-level thread does not block.

3.6 Interoperating with C

While I/O operations are the classical example of operations that block kernel-level threads, the non-blocking challenge extends to all blocking system-calls. The POSIX standard states [42, § 2.9.1]:

> All functions defined by this volume of POSIX.1-2017 shall be thread-safe, except that the following functions need not be thread-safe. ... (list of 70+ excluded functions)

Only UNIX `man` pages identify whether a library function is thread-safe, and hence, may block on a pthreads lock or system call; hence, interoperability with UNIX library functions is a challenge for an M:N threading model.

Languages like Go and Java, which have strict interoperability with C[101, 52], can control operations in C by "sandboxing" them, *e.g.*, a blocking function may be delegated to a kernel-level thread. Sandboxing may help towards guaranteeing that the kind of deadlock mentioned above does not occur.

[3] In this example, the deadlock could be resolved if the server sends unprompted messages to the client. However, this solution is neither general nor appropriate even in this simple case.

As mentioned in Section 1, C∀ is binary compatible with C and, as such, must support all C library functions. Furthermore, interoperability can happen at the function-call level, inline code, or C and C∀ translation units linked together. This fine-grained interoperability between C and C∀ has two consequences:

1. Precisely identifying blocking C calls is difficult.
2. Introducing safe-point code (see Go page 12) can have a significant impact on general performance.

Because of these consequences, this work does not attempt to "sandbox" calls to C. Therefore, it is possible calls to an unknown library function can block a kernel-level thread leading to deadlocks in C∀'s M:N threading model, which would not occur in a traditional 1:1 threading model. Since the blocking call is not known to the runtime, it is not necessarily possible to distinguish whether or not a deadlock occurs. Currently, all M:N thread systems interacting with UNIX without sandboxing suffer from this problem but manage to work very well in the majority of applications. Therefore, a complete solution to this problem is outside the scope of this thesis. [4] Chapter 5 discusses the interoperability with C chosen and used for the evaluation in Chapter 8.

[4] C∀ does provide a pthreads emulation, so any library function using embedded pthreads locks is redirected to C∀ user-level locks. This capability further reduces the chances of blocking a kernel-level thread.

Part II

Design

Chapter 4

Scheduling Core

Before discussing scheduling in general, where it is important to address systems that are changing states, this document discusses scheduling in a somewhat ideal scenario, where the system has reached a steady state. For this purpose, a steady state is loosely defined as a state where there are always threads ready to run and the system has the resources necessary to accomplish the work, *e.g.*, enough workers. In short, the system is neither overloaded nor underloaded.

It is important to discuss the steady state first because it is the easiest case to handle and, relatedly, the case in which the best performance is to be expected. As such, when the system is either overloaded or underloaded, a common approach is to try to adapt the system to this new load and return to the steady state, *e.g.*, by adding or removing workers. Therefore, flaws in scheduling the steady state tend to be pervasive in all states.

4.1 Design Goals

As with most of the design decisions behind C∀, an important goal is to match the expectation of the programmer according to their execution mental model. To match expectations, the design must offer the programmer sufficient guarantees so that, as long as they respect the execution mental model, the system also respects this model.

For threading, a simple and common execution mental model is the "ideal multitasking CPU":

> [The] "ideal multi-tasking CPU" is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at [an equal fraction of the] speed. For example: if there are 2 running tasks, then it runs each at 50% physical power — i.e., actually in parallel. (Linux CFS[4])

Applied to threads, this model states that every ready thread immediately runs in parallel with all other ready threads. While a strict implementation of this model is not feasible, programmers still have expectations about scheduling that come from this model.

In general, the expectation at the centre of this model is that ready threads do not interfere with each other but simply share the hardware. This assumption makes it easier to reason about

threading because ready threads can be thought of in isolation and the effect of the scheduler can be virtually ignored. This expectation of thread independence means the scheduler is expected to offer two features:

1. A fairness guarantee: a thread that is ready to run is not prevented by another thread indefinitely, *i.e.*, starvation freedom. This is discussed further in the next section.
2. A performance goal: given a thread that wants to start running, other threads wanting to do the same do not interfere with it.

The performance goal, the lack of interference among threads, is only desired up to a point. Ideally, the cost of running and blocking should be constant regardless of contention, but the goal is considered satisfied if the cost is not *too high* with or without contention. How much is an acceptable cost is obviously highly variable. For this document, the performance experimentation attempts to show the cost of scheduling is at worst equivalent to existing algorithms used in popular languages. This demonstration can be made by comparing applications built in C∀ to applications built with other languages or other models. Recall programmer expectation is that the impact of the scheduler can be ignored. Therefore, if the cost of scheduling is competitive with other popular languages, the goal is considered satisfied. More precisely the scheduler should be:

- As fast as other schedulers without any fairness guarantee.
- Faster than other schedulers that have equal or stronger fairness guarantees.

4.1.1 Fairness Goals

For this work, fairness is considered to have two strongly related requirements:

Starvation freedom means as long as at least one processor continues to dequeue threads, all ready threads should be able to run eventually, *i.e.*, eventual progress. Starvation freedom can be bounded or unbounded. In the bounded case, all threads should be able to run within a fix bound, relative to its own enqueue. Whereas unbounded starvation freedom only requires the thread to eventually run. The C∀ scheduler aims to guarantee unbounded starvation freedom. In any running system, a processor can stop dequeuing threads if it starts running a thread that never blocks. Without preemption, traditional work-stealing schedulers do not have starvation freedom, bounded or unbounded. Now, this requirement raises the question, what about preemption? Generally speaking, preemption happens on the timescale of several milliseconds, which brings us to the next requirement: "fast" load balancing.

Fast load balancing means that while eventual progress is guaranteed, it is important to mention on which timescale this progress is expected to happen. Indeed, while a scheduler with bounded starvation freedom is beyond the scope of this work, offering a good expected bound in the mathematical sense [105] is desirable. The expected bound on starvation freedom should be tighter than what preemption normally allows. For interactive applications that need to run at 60, 90 or 120 frames per second, threads having to wait milliseconds to run are effectively starved. Therefore load-balancing should be done at a faster pace: one that is expected to detect starvation at the microsecond scale.

4.1.2 Fairness vs Scheduler Locality

An important performance factor in modern architectures is cache locality. Waiting for data at lower levels or not present in the cache can have a major impact on performance. Having multiple hardware threads writing to the same cache lines also leads to cache lines that must be waited on. It is therefore preferable to divide data among each hardware thread[1].

For a scheduler, having good locality, *i.e.*, having the data local to each hardware thread, generally conflicts with fairness. Indeed, good locality often requires avoiding the movement of cache lines, while fairness requires dynamically moving a thread, and as a consequence cache lines, to a hardware thread that is currently available. Note that this section discusses *internal locality*, *i.e.*, the locality of the data used by the scheduler, versus *external locality*, *i.e.*, how scheduling affects the locality of the application's data. External locality is a much more complicated subject and is discussed in the next section.

However, I claim that in practice it is possible to strike a balance between fairness and performance because these requirements do not necessarily overlap temporally. Figure 4.1 shows a visual representation of this effect. As mentioned, some unfairness is acceptable; for example, once the bounded starvation guarantee is met, additional fairness will not satisfy it *more*. Inversely, once a thread's data is evicted from cache, its locality cannot worsen. Therefore it is desirable to have an algorithm that prioritizes cache locality as long as the fairness guarantee is also satisfied.

4.1.3 Performance Challenges

While there exists a multitude of potential scheduling algorithms, they generally always have to contend with the same performance challenges. Since these challenges are recurring themes in the design of a scheduler it is relevant to describe them here before looking at the scheduler's design.

Latency

The most basic performance metric of a scheduler is scheduling latency. This measures the how long it takes for a thread to run once scheduled, including the cost of scheduling itself. This measure include both the sequential cost of the operation itself, both also the scalability.

Scalability

Given a large number of processors and an even larger number of threads, scalability measures how fast processors can enqueue and dequeue threads relative to the available parallelism. One could expect that doubling the number of processors would double the rate at which threads are dequeued, but contention on the internal data structure of the scheduler can diminish the improvements. While the ready queue itself can be sharded to alleviate the main source of contention,

[1]This partitioning can be an explicit division up front or using data structures where different hardware threads are naturally routed to different cache lines.

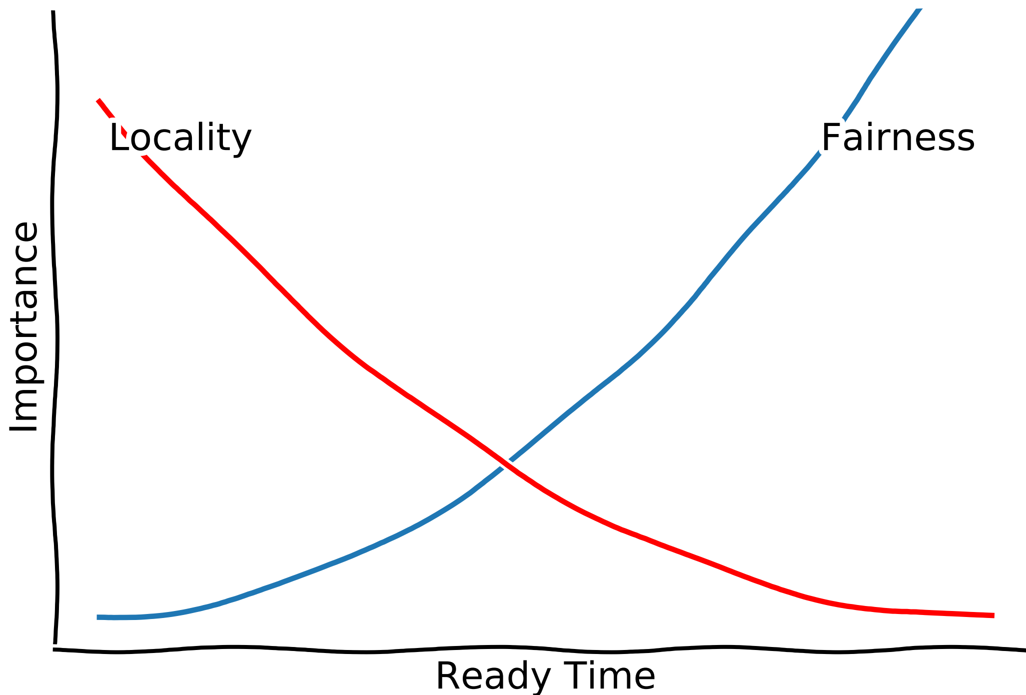


Figure 4.1: Rule of thumb Fairness vs Locality graph

The importance of Fairness and Locality while a ready thread awaits running is shown as the time the ready thread waits increases (Ready Time) the chances that its data is still in cache decreases (Locality). At the same time, the need for fairness increases since other threads may have the chance to run many times, breaking the fairness model. Since the actual values and curves of this graph can be highly variable, the graph is an idealized representation of the two opposing goals.

auxiliary scheduling features, *e.g.*, counting ready threads, can also be sources of contention. In the Chapter 7, scalability is measured as $\#procs \times \frac{ns}{ops}$, *i.e.*, number of processors times total time over total operations. Since the total number of operation should scale with the number of processors, this gives a measure how much each additional processor affects the other processors.

Migration Cost

Another important source of scheduling latency is migration. A thread migrates if it executes on two different processors consecutively, which is the process discussed in 4.1.2. Migrations can have many different causes, but in certain programs, it can be impossible to limit migration. Chapter 7 has a benchmark where any thread can potentially unblock any other thread, which can lead to threads migrating frequently. Hence, it is important to design the internal data structures of the scheduler to limit any latency penalty from migrations.

4.2 Inspirations

In general, a naïve FIFO ready-queue does not scale with increased parallelism from hardware threads, resulting in decreased performance. The problem is a single point of contention when adding/removing threads. As is shown in the evaluation sections, most production schedulers do scale when adding hardware threads. The solution to this problem is to shard the ready queue: create multiple *sub-queues* forming the logical ready-queue. The sub-queues are accessed by multiple hardware threads without the need for communication.

Before going into the design of C∀'s scheduler, it is relevant to discuss two sharding solutions that served as the inspiration for the scheduler in this thesis.

4.2.1 Work-Stealing

As mentioned in 2.4, a popular sharding approach for the ready queue is work-stealing. In this approach, each processor has its own local sub-queue and processors only access each other's sub-queue if they run out of work on their local ready-queue. The interesting aspect of work stealing manifests itself in the steady-state scheduling case, *i.e.*, all processors have work and no load balancing is needed. In this case, work stealing is close to optimal scheduling latency: it can achieve perfect locality and have no contention. On the other hand, work-stealing only attempts to do load-balancing when a processor runs out of work. This means that the scheduler never balances unfair loads unless they result in a processor running out of work. Chapter 7 shows that, in pathological cases, work stealing can lead to unbounded starvation.

Based on these observations, the conclusion is that a *perfect* scheduler should behave similarly to work-stealing in the steady-state case, *i.e.*, avoid migrations in well balanced workloads, but load balance proactively when the need arises.

4.2.2 Relaxed-FIFO

A different scheduling approach is the "relaxed-FIFO" queue, as in [13]. This approach forgoes any ownership between processor and sub-queue, and simply creates a pool of sub-queues from which processors pick. Scheduling is performed as follows:

- All sub-queues are protected by TryLocks.
- Timestamps are added to each element of a sub-queue.
- A processor randomly tests sub-queues until it has acquired one or two queues, referred to as *randomly picking* or *randomly helping*.
- If two queues are acquired, the older of the two threads is dequeued from the front of the acquired queues.
- Otherwise, the thread from the single queue is dequeued.

The result is a queue that has both good scalability and sufficient fairness. The lack of ownership ensures that as long as one processor is still able to repeatedly dequeue elements, it is unlikely any element will delay longer than any other element. This guarantee contrasts with work-stealing, where a processor with a long sub-queue results in unfairness for its threads in comparison to a

processor with a short sub-queue. This unfairness persists until a processor runs out of work and steals.

An important aspect of this scheme's fairness approach is that the timestamps make it possible to evaluate how long elements have been in the queue. However, processors eagerly search for these older elements instead of focusing on specific queues, which negatively affects locality.

While this scheme has good fairness, its performance can be improved. Wide sharding is generally desired, *e.g.*, at least 4 queues per processor, and randomly picking non-empty queues is difficult when there are few ready threads. The next sections describe improvements I made to this existing algorithm. However, ultimately the "relaxed-FIFO" queue is not used as the basis of the C∀ scheduler.

4.3 Relaxed-FIFO++

The inherent fairness and decent performance with many threads make the relaxed-FIFO queue a good candidate to form the basis of a new scheduler. The problem case is workloads where the number of threads is barely greater than the number of processors. In these situations, the wide sharding of the ready queue means most of its sub-queues are empty. Furthermore, the non-empty sub-queues are unlikely to hold more than one item. The consequence is that a random dequeue operation is likely to pick an empty sub-queue, resulting in an unbounded number of selections. This state is generally unstable: each sub-queue is likely to frequently toggle between being empty and nonempty. Indeed, when the number of threads is *equal* to the number of processors, every pop operation is expected to empty a sub-queue and every push is expected to add to an empty sub-queue. In the worst case, a check of the sub-queues sees all are empty or full.

As this is the most obvious challenge, it is worth addressing first. The seemingly obvious solution is to supplement each sharded sub-queue with data that indicates whether the queue is empty/nonempty. This simplifies finding nonempty queues, *i.e.*, ready threads. The sharded data can be organized in different forms, *e.g.*, a bitmask or a binary tree that tracks the nonempty sub-queues, using a bit or a node per sub-queue, respectively. Specifically, many modern architectures have powerful bitmask manipulation instructions, and, searching a binary tree has good Big-O complexity. However, precisely tracking nonempty sub-queues is problematic. The reason is that the sub-queues are initially sharded with a width presumably chosen to avoid contention. However, tracking which ready queue is nonempty is only useful if the tracking data is dense, *i.e.*, tracks whether multiple sub-queues are empty. Otherwise, it does not provide useful information because reading this new data structure risks being as costly as simply picking a sub-queue at random. But if the tracking mechanism *is* denser than the shared sub-queues, then constant updates invariably create a new source of contention. Early experiments with this approach showed that randomly picking, even with low success rates, is often faster than bit manipulations or tree walks.

The exception to this rule is using local tracking. If each processor locally keeps track of empty sub-queues, then this can be done with a very dense data structure without introducing a new source of contention. However, the consequence of local tracking is that the information is incomplete. Each processor is only aware of the last state it saw about each sub-queue so this

information quickly becomes stale. Even on systems with low hardware thread count, *e.g.*, 4 or 8, this approach can quickly lead to the local information being no better than the random pick. This result is due in part to the cost of maintaining information and its poor quality.

However, using a very low-cost but inaccurate approach for local tracking can still be beneficial. If the local tracking is no more costly than a random pick, then *any* improvement to the success rate, however low it is, leads to a performance benefit. This suggests the following approach:

4.3.1 Dynamic Entropy

[63] The Relaxed-FIFO approach can be made to handle the case of mostly empty sub-queues by tweaking the Pseudo Random Number Generator that drives the random picking of sub-queues. The PRNG state can be seen as containing a list of all the future sub-queues that will be accessed. While this concept is not particularly useful on its own, the consequence is that if the PRNG algorithm can be run *backwards*, then the state also contains a list of all the sub-queues that were accessed. Luckily, bidirectional PRNG algorithms do exist, *e.g.*, some Linear Congruential Generators [99] support running the algorithm backwards while offering good quality and performance. This particular PRNG can be used as follows:

- Each processor maintains two PRNG states, referred to as F and B.
- When a processor attempts to dequeue a thread, it picks a sub-queue by running its B backwards.
- When a processor attempts to enqueue a thread, it runs its F forward picking a sub-queue to enqueue to. If the enqueue is successful, state of its B is overwritten with the content of its F.

The result is that each processor tends to dequeue threads that it has itself enqueued. When most sub-queues are empty, this technique increases the odds of finding threads at a very low cost, while also offering an improvement on locality in many cases.

My own tests showed this approach performs better than relaxed-FIFO in many cases. However, it is still not competitive with work-stealing algorithms. The fundamental problem is that the randomness limits how much locality the scheduler offers. This becomes problematic both because the scheduler is likely to get cache misses on internal data structures and because migrations become frequent. Therefore, the attempt to modify the relaxed-FIFO algorithm to behave more like work stealing did not pan out. The alternative is to do it the other way around.

4.4 Work Stealing++

To add stronger fairness guarantees to work stealing a few changes are needed. First, the relaxed-FIFO algorithm has fundamentally better fairness because each processor always monitors all sub-queues. Therefore, the work-stealing algorithm must be prepended with some monitoring. Before attempting to dequeue from a processor's sub-queue, the processor must make some effort to ensure other sub-queues are not being neglected. To make this possible, processors must be able to determine which thread has been on the ready queue the longest. Second,

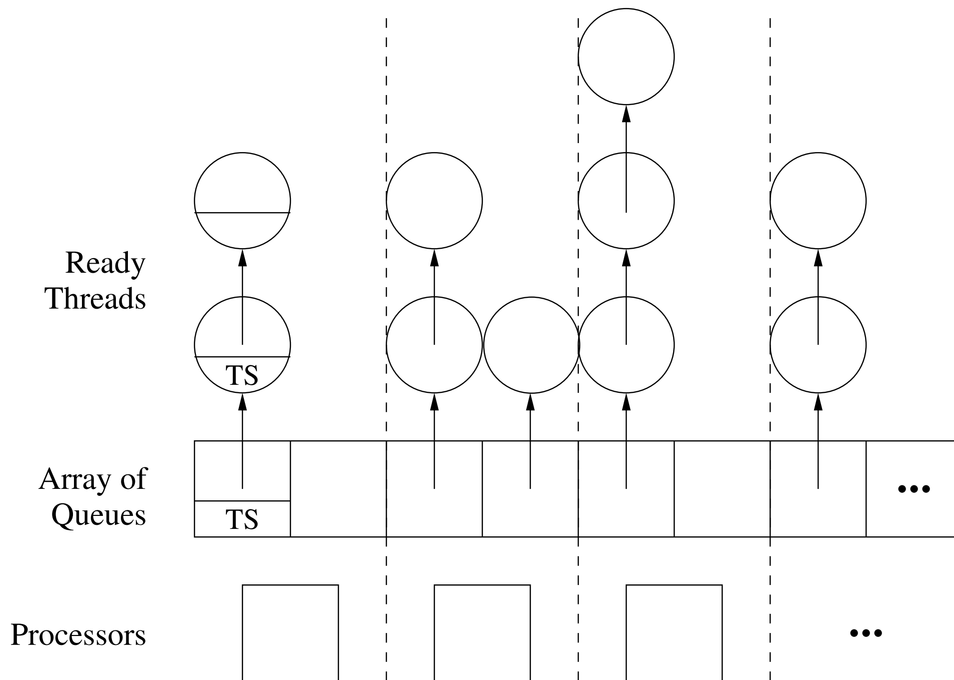


Figure 4.2: Base C∀ design
It uses a pool of sub-queues, with a sharding of two sub-queue per processor. Each processor can access all of the sub-queues. Each thread is timestamped when enqueued.

the relaxed-FIFO approach uses timestamps, denoted TS, for each thread to make this possible. Theses timestamps can be added to work stealing.

Figure 4.2 shows the algorithm structure. This structure is similar to classic work-stealing except the sub-queues are placed in an array so processors can access them in constant time. Sharding can be adjusted based on contention. As an optimization, the timestamp of a thread is stored in the thread in front of it, so the first TS is in the array and the last thread has no TS. This organization keeps the highly accessed front TSs directly in the array. When a processor attempts to dequeue a thread, it first picks a random remote sub-queue and compares its timestamp to the timestamps of its local sub-queue(s). The oldest waiting of the compared threads is dequeued. In this document, picking from a remote sub-queue in this fashion is referred to as “helping”.

The timestamps are measured using the CPU’s hardware timestamp counters [109]. These provide a 64-bit counter that tracks the number of cycles since the CPU was powered on. Assuming the CPU runs at less than 5 GHz, this means that the 64-bit counter takes over a century before overflowing. This is true even on 32-bit CPUs, where the counter is generally still 64-bit. However, on many architectures, the instructions to read the counter do not have any particular ordering guarantees. Since the counter does not depend on any data in the cpu pipeline, this means there is significant flexibility for the instruction to be read out of order, which limites the accuracy to a window of code. Finally, another issue that can come up with timestamp counters is synchronization between hardware threads. This appears to be mostly a historical concern, as recent CPU offer more synchronization guarantees. For example, Intel supports ”Invariant TSC” [46, § 17.15.1] which is guaranteed to be synchronized across hardware threads.

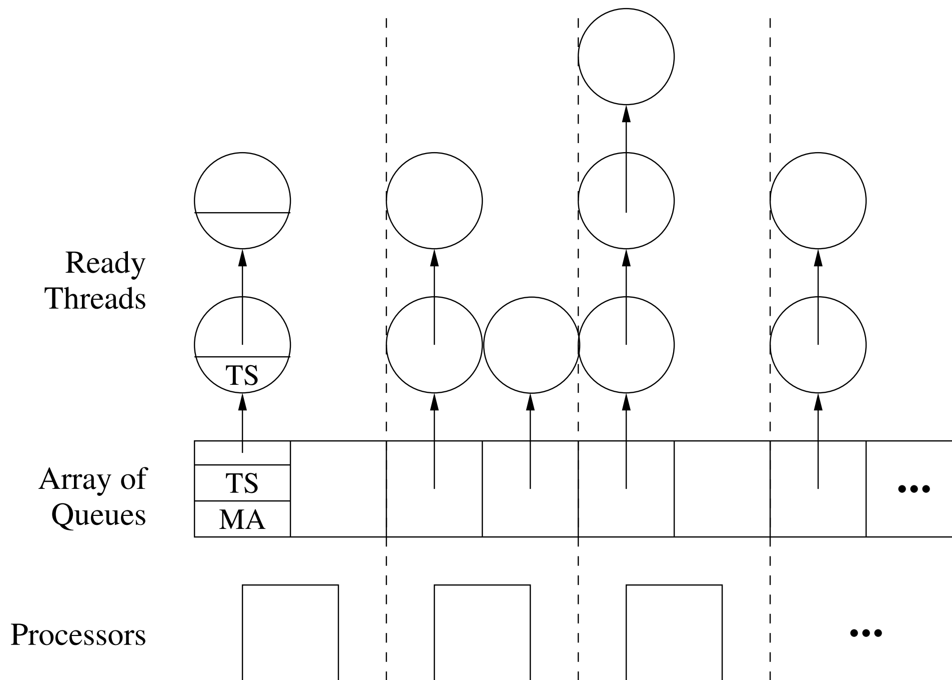


Figure 4.3: C∀ design with Moving Average
A moving average is added to each sub-queue.

However, this naïve implementation has performance problems. First, it is necessary to avoid helping when it does not improve fairness. Random effects like cache misses and preemption can add unpredictable but short bursts of latency but do not warrant the cost of helping. These bursts can cause increased migrations, at which point this same locality problems as in the relaxed-FIFO approach start to appear.

A simple solution to this problem is to use an exponential moving average[104] (MA) instead of a raw timestamp, as shown in Figure 4.3. Note that this is more complex than it can appear because the thread at the head of a sub-queue is still waiting, so its wait time has not ended. Therefore, the exponential moving average is an average of how long each dequeued thread has waited. To compare sub-queues, the timestamp at the head must be compared to the current time, yielding the best-case wait time for the thread at the head of the queue. This new waiting is averaged with the stored average. To further limit migrations, a bias can be added to a local sub-queue, where a remote sub-queue is helped only if its moving average is more than X times the local sub-queue's average. Tests for this approach indicate the precise values for the weight of the moving average and the bias are not important, *i.e.*, weights and biases of similar *magnitudes* have similar effects.

With these additions to work stealing, scheduling can satisfy the starvation freedom guarantee while suffering much less from unnecessary migrations than the relaxed-FIFO approach. Unfortunately, the work to achieve fairness has a performance cost, especially when the workload is inherently fair, and hence, there is only short-term unfairness or no starvation. The problem is that the constant polling, *i.e.*, reads, of remote sub-queues generally entails cache misses because the TSs are constantly being updated. To make things worse, remote sub-queues that are very

active, *i.e.*, threads are frequently enqueued and dequeued from them, lead to higher chances that polling will incur a cache-miss. Conversely, the active sub-queues do not benefit much from helping since starvation is already a non-issue. This puts this algorithm in the awkward situation of paying for a largely unnecessary cost. The good news is that this problem can be mitigated.

4.4.1 Redundant Timestamps

The problem with polling remote sub-queues is that correctness is critical. There must be a consensus among processors on which sub-queues hold which threads, as the threads are in constant motion. Furthermore, since timestamps are used for fairness, it is critical that the oldest threads eventually be recognized as such. However, when deciding if a remote sub-queue is worth polling, correctness is less of a problem. Since the only requirement is that a sub-queue is eventually polled, some data staleness is acceptable. This leads to a situation where stale timestamps are only problematic in some cases. Furthermore, stale timestamps can be desirable since lower freshness requirements mean fewer cache invalidations.

Figure 4.4 shows a solution with a second array containing a copy of the timestamps and average. This copy is updated *after* the sub-queue's critical sections using relaxed atomics. Processors now check if polling is needed by comparing the copy of the remote timestamp instead of the actual timestamp. The result is that since there is no fencing, the writes can be buffered in the hardware and cause fewer cache invalidations.

The correctness argument is somewhat subtle. The data used for deciding whether or not to poll a queue can be stale as long as it does not cause starvation. Therefore, it is acceptable if stale data makes queues appear older than they are, but appearing fresher can be a problem. For the timestamps, this means it is acceptable to miss writes to the timestamp since they make the head thread look older. For the moving average, as long as the operations are just atomic reads/writes, the average is guaranteed to yield a value that is between the oldest and newest values written. Therefore, this unprotected read of the timestamp and average satisfies the limited correctness that is required.

With redundant timestamps, this scheduling algorithm achieves both the fairness and performance requirements on most machines. The problem is that the cost of polling and helping is not necessarily consistent across each hardware thread. For example on machines with multiple CPUs, cache misses can be satisfied from the caches on the same (local) CPU, or by the caches on a different (remote) CPU. Cache misses satisfied by a remote CPU have significantly higher latency than from the local CPU. However, these delays are not specific to systems with multiple CPUs. Depending on the cache structure, cache misses can have different latency on the same CPU, *e.g.*, the AMD EPYC 7662 CPUs used in Chapter 7.

Figures 4.5 and 4.6 show two different cache topologies that highlight this difference. In Figure 4.5, all cache misses are either private to a CPU or shared with another CPU. This means that latency due to cache misses is fairly consistent. In contrast, in Figure 4.6, misses in the L2 cache can be satisfied by either instance of the L3 cache. However, the memory-access latency to the remote L3 is higher than the memory-access latency to the local L3. The impact of these different designs on this algorithm is that scheduling only scales well on architectures with the L3 cache shared across many hardware threads, similar to Figure 4.5, and less well on architectures

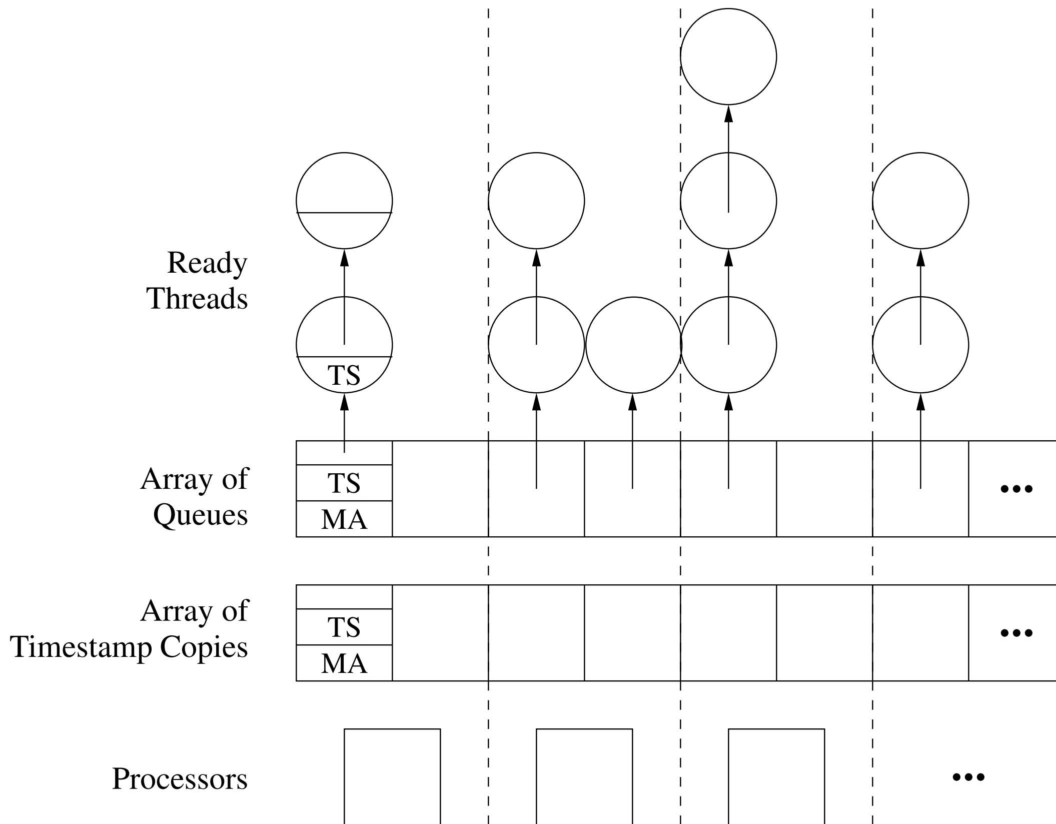


Figure 4.4: C∀ design with Redundant Timestamps
This design uses an array containing a copy of the timestamps. These timestamps are written-to with relaxed atomics, so there is no order among concurrent memory accesses, leading to fewer cache invalidations.

with many L3 cache instances and less sharing, similar to Figure 4.6. Hence, as the number of L3 instances grows, so too does the chance that the random helping causes significant cache latency. The solution is for the scheduler to be aware of the cache topology.

4.4.2 Per CPU Sharding

Building a scheduler that is cache aware poses two main challenges: discovering the cache topology and matching processors to this cache structure. Unfortunately, there is no portable way to discover cache topology, and it is outside the scope of this thesis to solve this problem. This work uses the cache topology information from Linux's `/sys/devices/system/cpu` directory. This leaves the challenge of matching processors to cache structure, or more precisely, identifying which sub-queues of the ready queue are local to which subcomponents of the cache structure. Once a match is generated, the helping algorithm is changed to add bias so that processors more often help sub-queues local to the same cache substructure.[2]

[2]Note that like other biases mentioned in this section, the actual bias value does not appear to need precise tuning beyond the order of magnitude.

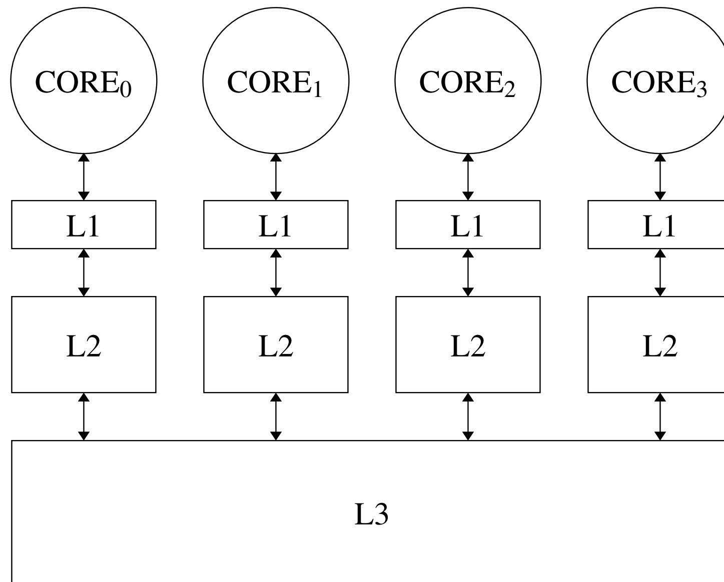


Figure 4.5: CPU design with wide L3 sharing
A CPU with 4 cores, where caches L1 and L2 are private to each core, and the L3 cache is shared across all cores.

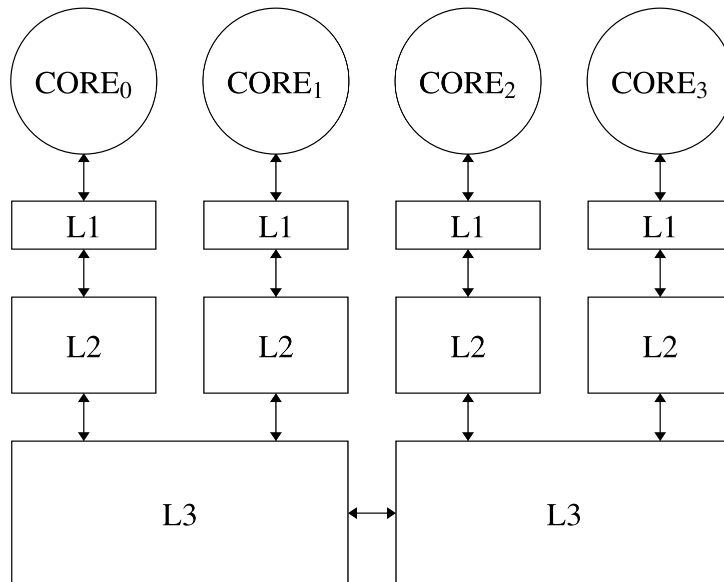


Figure 4.6: CPU design with a narrow L3 sharing
A CPU with 4 cores, where caches L1 and L2 are private to each core, and the L3 cache is shared across a pair of cores.

The simplest approach for mapping sub-queues to cache structure is to statically tie sub-queues to CPUs. Instead of having each sub-queue local to a specific processor, the system is initialized with sub-queues for each hardware hyperthread/core up front. Then processors dequeue and enqueue by first asking which CPU id they are executing on, to identify which sub-queues are the local ones. Processors can get the CPU id from `sched_getcpu` or `librseq`.

This approach solves the performance problems on systems with topologies with narrow L3 caches, similar to Figure 4.6. However, it can still cause some subtle fairness problems in systems with few processors and many hardware threads. In this case, the large number of sub-queues and the bias against sub-queues tied to different cache substructures make it unlikely that every sub-queue is picked. To make things worse, the small number of processors means that few helping attempts are made. This combination of low selection and few helping attempts allow a thread to become stranded on a sub-queue for a long time until it gets randomly helped. On a system with 2 processors, 256 hardware threads, and a 100:1 bias, it can take multiple seconds for a thread to get dequeued from a remote queue. In this scenario, where each processor attempts to help on 50% of dequeues, the probability that a remote sub-queue gets help is $\frac{1}{51200}$ and follows a geometric distribution. Therefore the probability of the remote sub-queue gets help within the next 100'000 dequeues is only 85%. Assuming dequeues happen every 100ns, there is still 15% chance a thread could starve for more than 10ms and a 1% chance the thread starves for 33.33ms, the maximum latency tolerated for interactive applications. If few hardware threads share each cache instance, the probability that a thread is on a remote sub-queue becomes high. Therefore, a more dynamic match of sub-queues to cache instances is needed.

4.4.3 Topological Work Stealing

The approach used in the C∀ scheduler is to have per-processor sub-queues, but have an explicit data structure to track which cache substructure each sub-queue is tied to. This tracking requires some finesse, because reading this data structure must lead to fewer cache misses than not having the data structure in the first place. A key element, however, is that, like the timestamps for helping, reading the cache instance mapping only needs to give the correct result *often enough*. Therefore the algorithm can be built as follows: before enqueueing or dequeuing a thread, a processor queries the CPU id and the corresponding cache instance. Since sub-queues are tied to processors, a processor can then update the cache instance mapped to the local sub-queue(s). To avoid unnecessary cache line invalidation, the map is only written-to if the mapping changes.

This scheduler is used in the remainder of the thesis for managing CPU execution, but additional scheduling is needed to handle long-term blocking and unblocking, such as I/O.

Chapter 5

User Level I/O

As mentioned in Section 3.5, user-level I/O requires multiplexing the I/O operations of many threads onto fewer processors using asynchronous I/O operations. I/O operations, among others, generally block the kernel-level thread when the operation needs to wait for unavailable resources. When using user-level threading, this results in the processor blocking rather than the thread, hindering parallelism and potentially causing deadlocks (see Chapter 3.5). Different operating systems offer various forms of asynchronous operations and, as mentioned in Chapter 1, this work is exclusively focused on the Linux operating system.

5.1 Kernel Interface

Since this work fundamentally depends on operating-system support, the first step of this design is to discuss the available interfaces and pick one (or more) as the foundation for the non-blocking I/O subsystem in this work.

5.1.1 `O_NONBLOCK`

In Linux, files can be opened with the flag `O_NONBLOCK` [65] (or `SO_NONBLOCK` [9], the equivalent for sockets) to use the file descriptors in "nonblocking mode". In this mode, "Neither the `open()` nor any subsequent I/O operations on the [opened file descriptor] will cause the calling process to wait" [65]. This feature can be used as the foundation for the non-blocking I/O subsystem. However, for the subsystem to know when an I/O operation completes, `O_NONBLOCK` must be used in conjunction with a system call that monitors when a file descriptor becomes ready, *i.e.*, the next I/O operation on it does not cause the process to wait.[1] This mechanism is also crucial in determining when all threads are blocked and the application kernel-level threads can now block.

There are three options to monitor file descriptors (FD) in Linux:[2] `select` [77], `poll` [69] and `epoll` [34]. All three of these options offer a system call that blocks a kernel-level thread

[1] In this context, ready means *some* operation can be performed without blocking. It does not mean an operation returning `EAGAIN` succeeds on the next try. For example, a ready read may only return a subset of requested bytes and the read must be issued again for the remaining bytes, at which point it may return `EAGAIN`.

[2] For simplicity, this section omits `pselect` and `ppoll`. The difference between these system calls and `select` and `poll`, respectively, is not relevant for this discussion.

until at least one of many file descriptors becomes ready. The group of file descriptors being waited on is called the *interest set*.

`select` is the oldest of these options, and takes as input a contiguous array of bits, where each bit represents a file descriptor of interest. Hence, the array length must be as long as the largest FD currently of interest. On return, it outputs the set modified in-place to identify which of the file descriptors changed state. This destructive change means selecting in a loop requires re-initializing the array for each iteration. Another limitation of `select` is that calls from different kernel-level threads sharing FDs are independent. Hence, if one kernel-level thread is managing the select calls, other threads can only add/remove to/from the manager's interest set through synchronized calls to update the interest set. However, these changes are only reflected when the manager makes its next call to `select`. Note, it is possible for the manager thread to never unblock if its current interest set never changes, *e.g.*, the sockets/pipes/TTYs it is waiting on never get data again. Often the I/O manager has a timeout, polls, or is sent a signal on changes to mitigate this problem.

`poll` is the next oldest option, and takes as input an array of structures containing the FD numbers rather than their position in an array of bits, allowing a more compact input for interest sets that contain widely spaced FDs. For small interest sets with densely packed FDs, the `select` bit mask can take less storage, and hence, copy less information into the kernel. However, `poll` is non-destructive, so the array of structures does not have to be re-initialized on every call. Like `select`, `poll` suffers from the limitation that the interest set cannot be changed by other kernel-level threads, while a manager thread is blocked in `poll`.

`epoll` follows after `poll`, and places the interest set in the kernel rather than the application, where it is managed by an internal kernel-level thread. There are two separate functions: one to add to the interest set and another to check for FDs with state changes. This dynamic capability is accomplished by creating an *epoll instance* with a persistent interest set, which is used across multiple calls. As the interest set is augmented, the changes become implicitly part of the interest set for a blocked manager kernel-level thread. This capability significantly reduces synchronization between kernel-level threads and the manager calling `epoll`.

However, all three of these I/O systems have limitations. The `man` page for `O_NONBLOCK` mentions that "[`O_NONBLOCK`] has no effect for regular files and block devices", which means none of these three system calls are viable multiplexing strategies for these types of I/O operations. Furthermore, TTYs (FDs connect to a standard input and output) can also be tricky to use since they can take different forms based on how the command is executed. For example, `epoll` rejects FDs pointing to regular files or block devices, which includes `stdin` when using shell redirections [71, § 3.6], but does not reject shell pipelines [71, § 3.2.3], which includes pipelines into `stdin`. Finally, none of these are useful solutions for multiplexing I/O operations that do not have a corresponding file descriptor and can be awkward for operations using multiple file descriptors.

5.1.2 POSIX asynchronous I/O (AIO)

An alternative to `O_NONBLOCK` is the AIO interface. Using AIO, programmers can enqueue operations which are to be performed asynchronously by the kernel. The kernel can communicate completions of these operations in three ways: it can spawn a new kernel-level thread; send a Linux signal; or userspace can poll for completion of one or more operations. Spawning a new kernel-level thread is not consistent with working at the user-level thread level, but Section 5.1.4 discusses a related solution. Signals and their associated interrupt handlers can also lead to fairly complicated interactions between subsystems, and they have a non-trivial cost. This leaves a single option: polling for completion—this is similar to the previously discussed system calls. While AIO only supports read and write operations to file descriptors; it does not have the same limitations as `O_NONBLOCK`, *i.e.*, the file descriptors can be regular files or block devices. AIO also supports batching multiple operations in a single system call.

AIO offers two different approaches to polling: `aio_error` can be used as a spinning form of polling, returning `EINPROGRESS` until the operation is completed, while `aio_suspend` can be used similarly to `select`, `poll` or `epoll`, to wait until one or more requests have been completed. Asynchronous interfaces normally handle more of the complexity than retry-based interfaces, which is convenient for I/O multiplexing. However, even if AIO requests can be submitted concurrently, `aio_suspend` suffers from the same limitation as `select` and `poll`: the interest set cannot be dynamically changed while a call to `aio_suspend` is in progress. AIO also suffers from the limitation of specifying which requests have been completed, *i.e.*, programmers have to poll each request in the interest set using `aio_error` to identify the completed requests. This limitation means that, like `select` and `poll` but not `epoll`, the time needed to examine polling results increases based on the total number of requests monitored, not the number of completed requests. Finally, AIO does not seem to be a popular interface, which I believe is due in part to this poor polling interface. Linus Torvalds talks about this interface as follows:

> AIO is a horrible ad-hoc design, with the main excuse being “other, less gifted people, made that design, and we are implementing it for compatibility because database people - who seldom have any shred of taste - actually use it”.
>
> But AIO was always really really ugly.
>
> – Linus Torvalds [90]

Interestingly, in this e-mail, Linus goes on to describe “a true *asynchronous system call* interface” that does “[an] arbitrary system call X with arguments A, B, C, D asynchronously using a kernel thread” in “some kind of arbitrary *queue up asynchronous system call* model”. This description is quite close to the interface described in the next section.

5.1.3 `io_uring`

A very recent addition to Linux, `io_uring` [18], is a framework that aims to solve many of the problems listed in the above interfaces. Like AIO, it represents I/O operations as entries

added to a queue. But like `epoll`, new requests can be submitted while a blocking call waiting for requests to complete is already in progress. The `io_uring` interface uses two ring buffers (referred to simply as rings) at its core: a submit ring, to which programmers push I/O requests, and a completion ring, from which programmers poll for completion.

One of the big advantages over the prior interfaces is that `io_uring` also supports a much wider range of operations. In addition to supporting reads and writes to any file descriptor like AIO, it also supports other operations, like `open`, `close`, `fsync`, `accept`, `connect`, `send`, `recv`, `splice`, *etc.*

On top of these, `io_uring` adds many extras like avoiding copies between the kernel and user space using shared memory, allowing different mechanisms to communicate with device drivers, and supporting chains of requests, *i.e.*, requests that automatically trigger follow-up requests on completion.

5.1.4 Extra Kernel Threads

Finally, if the operating system does not offer a satisfactory form of asynchronous I/O operations, an ad hoc solution is to create a pool of kernel-level threads and delegate operations to it to avoid blocking processors, which is a compromise for multiplexing. In the worst case, where all threads are consistently blocking on I/O, it devolves into 1-to-1 threading. However, regardless of the frequency of I/O operations, it achieves the fundamental goal of not blocking processors when threads are ready to run. This approach is used by languages like Go [53], frameworks like libuv [56], and web servers like Apache [14] and NGINX [64], since it has the advantage that it can easily be used across multiple operating systems. This advantage is especially relevant for languages like Go, which offer a homogeneous API across all platforms. Contrast this to C, which has a very limited standard API for I/O, *e.g.*, the C standard library has no networking.

5.1.5 Discussion

These options effectively fall into two broad camps: waiting for I/O to be ready, versus waiting for I/O to complete. All operating systems that support asynchronous I/O must offer an interface along at least one of these lines, but the details vary drastically. For example, FreeBSD offers `kqueue` [7], which behaves similarly to `epoll`, but with some small quality of life improvements, while Windows (Win32) [83] offers "overlapped I/O", which handles submissions similarly to `O_NONBLOCK` with extra flags on the synchronous system call, but waits for completion events, similarly to `io_uring`.

For this project, I selected `io_uring`, in large part because of its generality. While `epoll` has been shown to be a good solution for socket I/O ([50]), `io_uring`'s transparent support for files, pipes, and more complex operations, like `splice` and `tee`, make it a better choice as the foundation for a general I/O subsystem.

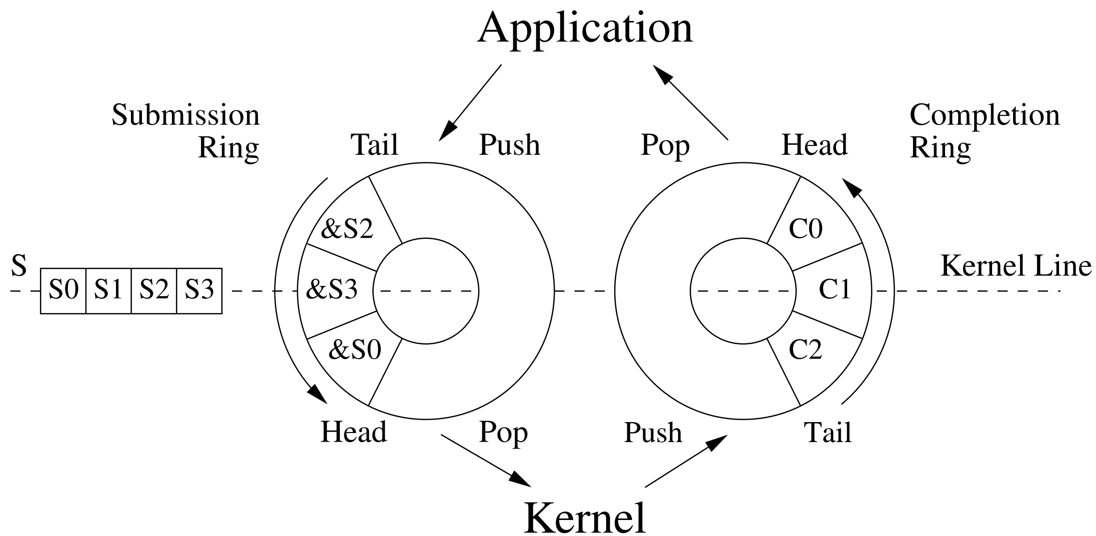


Figure 5.1: Overview of `io_uring`

Two ring buffers are used to communicate with the kernel, one for completions (right) and one for submissions (left). While the completion ring contains plain data, the submission ring contains only references. These references are indexes into an array (denoted *S*), which is created at the same time as the two rings and is also readable by the kernel.

5.2 Event-Engine

An event engine's responsibility is to use the kernel interface to multiplex many I/O operations onto few kernel-level threads. In concrete terms, this means threads enter the engine through an interface, the event engine then starts an operation and parks the calling threads, and then returns control to the processor. The parked threads are then rescheduled by the event engine once the desired operation has been completed.

5.2.1 `io_uring` in depth

Before going into details on the design of my event engine, more details on `io_uring` usage are presented, each important in the design of the engine. Figure 5.1 shows an overview of an `io_uring` instance. Two ring buffers are used to communicate with the kernel: one for submissions (left) and one for completions (right). The submission ring contains *Submit Queue Entries* (SQE), produced (appended) by the application when an operation starts and then consumed by the kernel. The completion ring contains *Completion Queue Entries* (CQE), produced (appended) by the kernel when an operation completes and then consumed by the application. The submission ring contains indexes into the SQE array (denoted *S* in the figure) containing entries describing the I/O operation to start; the completion ring contains entries for the completed I/O operation. Multiple `io_uring` instances can be created, in which case they each have a copy of the data structures in the figure.

New I/O operations are submitted to the kernel following 4 steps, which use the components shown in the figure.

1. An SQE is allocated from the pre-allocated array *S*. This array is created at the same time as the `io_uring` instance, is in kernel-locked memory visible by both the kernel and the application, and has a fixed size determined at creation. How these entries are allocated is not important for the functioning of `io_uring`; the only requirement is that no entry is reused before the kernel has consumed it.
2. The SQE is filled according to the desired operation. This step is straightforward. The only detail worth mentioning is that SQEs have a `user_data` field that must be filled to match submission and completion entries.
3. The SQE is submitted to the submission ring by appending the index of the SQE to the ring following regular ring buffer steps: `buffer[head] = item; head++`. Since the head is visible to the kernel, some memory barriers may be required to prevent the compiler from reordering these operations. Since the submission ring is a regular ring buffer, more than one SQE can be added at once and the head is updated only after all entries are updated. Note, SQE can be filled and submitted in any order, *e.g.*, in Figure 5.1 the submission order is S0, S3, S2. S1 has not been submitted.
4. The kernel is notified of the change to the ring using the system call `io_uring_enter`. The number of elements appended to the submission ring is passed as a parameter and the number of elements consumed is returned. The `io_uring` instance can be constructed so this step is not required, but this feature requires that the process have elevated privilege.

The completion side is simpler: applications call `io_uring_enter` with the flag `IORING_ENTER_GETEVENTS` to wait on a desired number of operations to complete. The same call can be used to both submit SQEs and wait for operations to complete. When operations do complete, the kernel appends a CQE to the completion ring and advances the head of the ring. Each CQE contains the result of the operation as well as a copy of the `user_data` field of the SQE that triggered the operation. The `io_uring_enter` system call is only needed if the application wants to block waiting for operations to complete or to flush the submission ring. `io_uring` supports option `IORING_SETUP_SQPOLL` at creation, which can remove the need for the system call for submissions.

The `io_uring_enter` system call is protected by a lock inside the kernel. This protection means that concurrent calls to `io_uring_enter` using the same instance are possible, but there is no performance gained from parallel calls to `io_uring_enter`. It is possible to do the first three submission steps in parallel; however, doing so requires careful synchronization.

`io_uring` also introduces constraints on the number of simultaneous operations that can be "in flight". First, SQEs are allocated from a fixed-size array, meaning that there is a hard limit to how many SQEs can be submitted at once. Second, the `io_uring_enter` system call can fail because "The kernel [...] ran out of resources to handle [a request]" or "The application is attempting to overcommit the number of requests it can have pending.". This restriction means I/O request bursts may have to be subdivided and submitted in chunks at a later time.

An important detail to keep in mind is that just like "The cloud is just someone else's computer" [62], asynchronous operations are just operations using someone else's threads. Indeed, asynchronous operations can require computation time to complete, which means that if this time is not taken from the thread that triggered the asynchronous operation, it must be taken from some other threads. In this case, the `io_uring` operations that cannot be handled directly in the

system call must be delegated to some other kernel-level thread. To this end, io_uring maintains multiple kernel-level threads inside the kernel that are not exposed to the user. Three kinds of operations that can need the kernel-level threads are:

Operations using IOSQE_ASYNC. This is a straightforward case, users can explicitly set the IOSQE_ASYNC flag on an SQE to specify that it *must* be delegated to a different kernel-level thread.

Bounded operations. This is also a fairly simple case. As mentioned earlier in this chapter, [O_NONBLOCK] has no effect for regular files and block devices. Therefore, io_uring handles this case by delegating operations on regular files and block devices. In fact, io_uring maintains a pool of kernel-level threads dedicated to these operations, which are referred to as *bounded workers*.

Unbounded operations that must be retried. While operations like reads on sockets can return EAGAIN instead of blocking the kernel-level thread, in the case these operations return EAGAIN they must be retried by io_uring once the data is available on the socket. Since this retry cannot necessarily be done in the system call, *i.e.*, using the application's kernel-level thread, io_uring must delegate these calls to kernel-level threads in the kernel. io_uring maintains a separate pool for these operations. The kernel-level threads in this pool are referred to as *unbounded workers*. Once unbounded operations are ready to be retried, one of the workers is woken up and it will handle the retry inside the kernel. Unbounded workers are also responsible for handling operations using IOSQE_ASYNC.

io_uring implicitly spawns and joins both the bounded and unbounded workers based on its evaluation of the needs of the workload. This effectively encapsulates the work that is needed when using epoll. Indeed, io_uring does not change Linux's underlying handling of I/O operations, it simply offers an asynchronous API on top of the existing system.

5.2.2 Multiplexing I/O: Submission

The submission side is the most complicated aspect of io_uring and the completion side effectively follows from the design decisions made on the submission side. While there is freedom in designing the submission side, there are some realities of io_uring that must be taken into account. It is possible to do the first steps of submission in parallel; however, the duration of the system call scales with the number of entries submitted. The consequence is that the amount of parallelism used to prepare submissions for the next system call is limited. Beyond this limit, the length of the system call is the throughput-limiting factor. I concluded from early experiments that preparing submissions seems to take almost as long as the system call itself, which means that with a single io_uring instance, there is no benefit in terms of I/O throughput to having more than two hardware threads. Therefore, the design of the submission engine must manage multiple instances of io_uring running in parallel, effectively sharding io_uring instances. Since completions are sent to the instance where requests were submitted, all instances with pending operations must be polled continuously[3]. Note that once an operation completes,

[3]As described in Chapter 6, this does not translate into high CPU usage.

there is nothing that ties it to the `io_uring` instance that handled it — nothing prevents a new operation, with for example the same file descriptor, from using a different `io_uring` instance.

A complicating aspect of submission is `io_uring`'s support for chains of operations, where the completion of an operation triggers the submission of the next operation on the link. SQEs forming a chain must be allocated from the same instance and must be contiguous in the Submission Ring (see Figure 5.1). The consequence of this feature is that filling SQEs can be arbitrarily complex, and therefore, users may need to run arbitrary code between allocation and submission. For this work, supporting chains is not a requirement of the C∀ I/O subsystem, but it is still valuable. Support for this feature can be fulfilled simply by supporting arbitrary user code between allocation and submission.

Similar to scheduling, sharding `io_uring` instances can be done privately, *i.e.*, one instance per processor, in decoupled pools, *i.e.*, a pool of processors using a pool of `io_uring` instances without one-to-one coupling between any given instance and any given processor, or some mix of the two. These three sharding approaches are analyzed.

Private Instances

The private approach creates one ring instance per processor, *i.e.*, one-to-one coupling. This alleviates the need for synchronization on the submissions, requiring only that threads are not time-sliced during submission steps. This requirement is the same as accessing `thread_local` variables, where a thread is accessing kernel-thread data, is time-sliced, and continues execution on another kernel thread but is now accessing the wrong data. This failure is the *serially reusable problem* [40]. Hence, allocated SQEs must be submitted to the same ring on the same processor, which effectively forces the application to submit SQEs in order of allocation.[4] From the subsystem's point of view, the allocation and submission are sequential, greatly simplifying both. In this design, allocation and submission form a partitioned ring buffer, as shown in Figure 5.2. Once added to the ring buffer, the attached processor has a significant amount of flexibility with regard to when to perform the system call. Possible options are: when the processor runs out of threads to run, after running a given number of threads, *etc*.

This approach has the advantage that it does not require much of the synchronization needed in a shared approach. However, this benefit means threads submitting I/O operations have less flexibility: they cannot park or yield, and several exceptional cases are handled poorly. Instances running out of SQEs cannot run threads wanting to do I/O operations. In this case, the I/O thread needs to be moved to a different processor, and the only current way of achieving this is to `yield()` hoping to be scheduled on a different processor with free SQEs, which is not guaranteed to ever occur.

A more involved version of this approach tries to solve these problems using a pattern called *helping*. Threads that cannot submit I/O operations, either because of an allocation failure or migration to a different processor between allocation and submission, create an I/O object and add it to a list of pending submissions per processor and a list of pending allocations, probably per cluster. While there is still a strong coupling between processors and `io_uring` instances,

[4] To remove this requirement, a thread needs the ability to "yield to a specific processor", *i.e.*, park with the guarantee it unparks on a specific processor, *i.e.*, the processor attached to the correct ring.

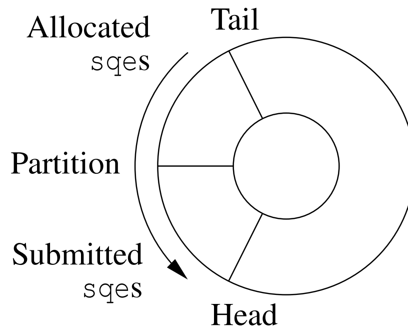


Figure 5.2: Partitioned ring buffer

Allocated SQEs are appended to the first partition. When submitting, the partition is advanced. The kernel considers the partition as the head of the ring.

these data structures allow moving threads to a specific processor, when the current processor cannot fulfill the I/O request.

Imagine a simple scenario with two threads on two processors, where one thread submits an I/O operation and then sets a flag, while the other thread spins until the flag is set. Assume both threads are running on the same processor, and the I/O thread is preempted between allocation and submission, moved to the second processor, and the original processor starts running the spinning thread. In this case, the helping solution has the I/O thread append an I/O object to the submission list of the first processor, where the allocation was made. No other processor can help the thread since `io_uring` instances are strongly coupled to processors. However, the I/O processor is unable to help because it is executing the spinning thread. This results in a deadlock. While this example is artificial, in the presence of many threads, this problem can arise "in the wild". Furthermore, this pattern is difficult to reliably detect and avoid. Once in this situation, the only escape is to interrupt the spinning thread, either directly or via some regular preemption, *e.g.*, time slicing. Having to interrupt threads for this purpose is costly, the latency can be large between interrupts, and the situation may be hard to detect. Interrupts are needed here entirely because the processor is tied to an instance it is not using. Therefore, a more satisfying solution is for the thread submitting the operation to notice that the instance is unused and simply go ahead and use it. This approach is presented shortly.

Public Instances

The public approach creates decoupled pools of `io_uring` instances and processors, *i.e.*, without one-to-one coupling. Threads attempting an I/O operation pick one of the available instances and submit the operation to that instance. Since there is no coupling between `io_uring` instances and processors in this approach, threads running on more than one processor can attempt to submit to the same instance concurrently. Because `io_uring` effectively sets the amount of sharding needed to avoid contention on its internal locks, performance in this approach is based on two aspects:

- The synchronization needed to submit does not induce more contention than `io_uring` already does.

- The scheme to route I/O requests to specific io_uring instances does not introduce contention. This aspect is very important because it comes into play before the sharding of instances, and as such, all hardware threads can contend on the routing algorithm.

Allocation in this scheme is fairly easy. Free SQEs, *i.e.*, SQEs that are not currently being used to represent a request, can be written-to safely, and have a field called user_data that the kernel only reads to copy to CQEs. Allocation also does not require ordering guarantees as all free SQEs are interchangeable. The only added complexity is that the number of SQEs is fixed, which means allocation can fail.

Allocation failures need to be pushed to a routing algorithm: threads attempting I/O operations must not be directed to io_uring instances without sufficient SQEs available. Furthermore, the routing algorithm should block operations upfront if none of the instances have available SQEs.

Once an SQE is allocated, threads insert the I/O request information and keep track of the SQE index and the instance it belongs to.

Once an SQE is filled in, it is added to the submission ring buffer, an operation that is not thread safe, and then the kernel must be notified using the io_uring_enter system call. The submission ring buffer is the same size as the pre-allocated SQE buffer, therefore pushing to the ring buffer cannot fail because it would mean an SQE multiple times in the ring buffer, which is undefined behaviour. However, as mentioned, the system call itself can fail with the expectation that it can be retried once some submitted operations are complete.

Since multiple SQEs can be submitted to the kernel at once, it is important to strike a balance between batching and latency. Operations that are ready to be submitted should be batched together in few system calls, but at the same time, operations should not be left pending for long periods before being submitted. Balancing submission can be handled by either designating one of the submitting threads as the thread responsible for the system call for the current batch of SQEs or by having some other party regularly submit all ready SQEs, *e.g.*, the poller thread mentioned later in this section.

Ideally, when multiple threads attempt to submit operations to the same io_uring instance, all requests should be batched together and one of the threads is designated to do the system call on behalf of the others, called the *submitter*. However, in practice, I/O requests must be handed promptly so there is a need to guarantee everything missed by the current submitter is seen by the next one. Indeed, as long as there is a "next" submitter, threads submitting new I/O requests can move on, knowing that some future system call includes their request. Once the system call is done, the submitter must also free SQEs so that the allocator can reuse them.

Finally, the completion side is much simpler since the io_uring system call enforces a natural synchronization point. Polling simply needs to regularly do the system call, go through the produced CQEs and communicate the result back to the originating threads. Since CQEs only own a signed 32-bit result, in addition to the copy of the user_data field, all that is needed to communicate the result is a simple future [97]. If the submission side does not designate submitters, polling can also submit all SQEs as it is polling events. A simple approach to polling is to allocate a user-level thread per io_uring instance and simply let the poller threads poll their respective instances when scheduled.

The big advantage of the pool of SQE instances approach is that it is fairly flexible. It does not impose restrictions on what threads submitting I/O operations can and cannot do between allocations and submissions. It also can gracefully handle running out of resources, SQEs or the kernel returning `EBUSY`. The downside to this approach is that many of the steps used for submitting need complex synchronization to work properly. The routing and allocation algorithm needs to keep track of which ring instances have available SQEs, block incoming requests if no instance is available, prevent barging if threads are already queued up waiting for SQEs and handle SQEs being freed. The submission side needs to safely append SQEs to the ring buffer, correctly handle chains, make sure no SQE is dropped or left pending forever, notify the allocation side when SQEs can be reused, and handle the kernel returning `EBUSY`. All this synchronization has a significant cost, compared to the private-instance approach which does not have synchronization costs in most cases.

Instance borrowing

Both of the prior approaches have undesirable aspects that stem from tight or loose coupling between `io_uring` and processors. The first approach suffers from tight coupling, causing problems when a processor does not benefit from the coupling. The second approach suffers from loose couplings, causing operations to have synchronization overhead, which tighter coupling avoids. When processors are continuously issuing I/O operations, tight coupling is valuable since it avoids synchronization costs. However, in unlikely failure cases or when processors are not using their instances, tight coupling is no longer advantageous. A compromise between these approaches is to allow tight coupling but have the option to revoke the coupling dynamically when failure cases arise. I call this approach *instance borrowing*.[5]

As mentioned later in this section, this approach is not ultimately used, but here is still an high-level outline of the algorithm. In this approach, each cluster, see Figure 3.1, owns a pool of `io_uring` instances managed by an *arbiter*. When a thread attempts to issue an I/O operation, it asks for an instance from the arbiter, and issues requests to that instance. This instance is now bound to the processor the thread is running on. This binding is kept until the arbiter decides to revoke it, taking back the instance and reverting the processor to its initial I/O state. This tight coupling means that synchronization can be minimal since only one processor can use the instance at a time, akin to the private instances approach. However, it differs in that revocation by the arbiter means this approach does not suffer from the deadlock scenario described above.

Arbitration is needed in the following cases:

1. The current processor does not hold an instance.
2. The current instance does not have sufficient SQEs to satisfy the request.
3. The current processor has a wrong instance. This happens if the submitting thread context-switched between allocation and submission: *external submissions*.

However, even when the arbiter is not directly needed, processors need to make sure that their instance ownership is not being revoked, which is accomplished by a lock-*less* handshake.[6] A

[5] While instance borrowing looks similar to work sharing and stealing, I think it is different enough to warrant a different verb to avoid confusion.

[6] Note the handshake is not lock-*free* [107] since it lacks the proper progress guarantee.

processor raises a local flag before using its borrowed instance and checks if the instance is marked as revoked or if the arbiter has raised its flag. If not, it proceeds, otherwise it delegates the operation to the arbiter. Once the operation is completed, the processor lowers its local flag.

Correspondingly, before revoking an instance, the arbiter marks the instance and then waits for the processor using it to lower its local flag. Only then does it reclaim the instance and potentially assign it to another processor.

The arbiter maintains four lists around which it makes its decisions:

1. A list of pending submissions.
2. A list of pending allocations.
3. A list of instances currently borrowed by processors.
4. A list of instances currently available.

External Submissions are handled by the arbiter by revoking the appropriate instance and adding the submission to the submission ring. However, there is no need to immediately revoke the instance. External submissions must simply be added to the ring before the next system call, *i.e.*, when the submission ring is flushed. This means whoever is responsible for the system call first checks whether the instance has any external submissions. If so, it asks the arbiter to revoke the instance and add the external submissions to the ring.

Pending Allocations are handled by the arbiter when it has available instances and can directly hand over the instance and satisfy the request. Otherwise, it must hold on to the list of threads until SQEs are made available again. This handling is more complex when an allocation requires multiple SQEs, since the arbiter must make a decision between satisfying requests in FIFO ordering or for fewer SQEs.

While an arbiter has the potential to solve many of the problems mentioned above, it also introduces a significant amount of complexity. Tracking which processors are borrowing which instances and which instances have SQEs available ends up adding a significant synchronization prelude to any I/O operation. Any submission must start with a handshake that pins the currently borrowed instance, if available. An attempt to allocate is then made, but the arbiter can concurrently be attempting to allocate from the same instance from a different hardware thread. Once the allocation is completed, the submission must check that the instance is still burrowed before attempting to flush. These synchronization steps turn out to have a similar cost to the multiple shared-instances approach. Furthermore, if the number of instances does not match the number of processors actively submitting I/O, the system can fall into a state where instances are constantly being revoked and end up cycling the processors, which leads to significant cache deterioration. For these reasons, this approach, which sounds promising on paper, does not improve on the private instance approach in practice.

5.3 Interface

The final part of the I/O subsystem is its interface. Multiple approaches can be offered to programmers, each with advantages and disadvantages. The new C∀ I/O subsystem can replace

the C runtime API or extend it, and in the latter case, the interface can go from very similar to vastly different. The following sections discuss some useful options, using `read` as an example. The standard Linux interface for C is:

```
ssize_t read(int fd, void *buf, size_t count);
```

5.3.1 Replacement

Replacing the C I/O subsystem is the more intrusive and draconian approach. The goal is to convince the compiler and linker to replace any calls to `read` by calls to the C∀ implementation instead of glibc's. This rerouting has the advantage of working transparently and supporting existing binaries without necessarily needing recompilation. It also offers a presumably well known and familiar API that C programmers can simply continue to work with. However, when using this approach, any and all calls to the C I/O subsystem, since using a mix of the C and C∀ I/O subsystems can easily lead to esoteric concurrency bugs. This approach was rejected as being laudable but infeasible.

5.3.2 Synchronous Extension

Another interface option is to offer an interface different in name only. In this approach, an alternative call is created for each supported system calls. For example:

```
ssize_t cfa_read(int fd, void *buf, size_t count);
```

The new `cfa_read` would have the same interface behaviour and guarantee as the `read` system call, but allow the runtime system to use user-level blocking instead of kernel-level blocking.

This approach is feasible and still familiar to C programmers. It comes with the caveat that any code attempting to use it must be modified, which is a problem considering the amount of existing legacy C binaries. However, it has the advantage of implementation simplicity. Finally, there is a certain irony to using a blocking synchronous interface for a feature often referred to as "non-blocking" I/O.

5.3.3 Asynchronous Extension

A fairly traditional way of providing asynchronous interactions is using a future mechanism [37], *e.g.*:

```
future(ssize_t) read(int fd, void *buf, size_t count);
```

where the generic `future` is fulfilled when the read completes, with the count of bytes actually read, which may be less than the number of bytes requested. The data read is placed in `buf`. The problem is that both the bytes count and data form the synchronization object, not just the bytes read. Hence, the buffer cannot be reused until the operation completes but the synchronization on the future does not enforce this. A classical asynchronous API is:

```
future([ssize_t, void *]) read(int fd, size_t count);
```

where the future tuple covers the components that require synchronization. However, this interface immediately introduces memory lifetime challenges since the call must effectively allocate a buffer to be returned. Because of the performance implications of this API, the first approach is considered preferable as it is more familiar to C programmers.

5.3.4 Direct `io_uring` Interface

The last interface directly exposes the underlying `io_uring` interface, *e.g.*:

```
array(SQE, want) cfa_io_allocate(int want);
void cfa_io_submit( const array(SQE, have) & );
```

where the generic `array` contains an array of SQEs with a size that may be less than the request. This offers more flexibility to users wanting to fully utilize all of the `io_uring` features. However, it is not the most user-friendly option. It obviously imposes a strong dependency between user code and `io_uring` but at the same time restricts users to usages that are compatible with how C∀ internally uses `io_uring`.

As of writting this document, C∀ offers both a synchronous extension and the first approach to the asynchronous extension:

```
ssize_t cfa_read(int fd, void *buf, size_t count);
future(ssize_t) async_read(int fd, void *buf, size_t count);
```

Chapter 6

Scheduling in practice

The scheduling algorithm described in Chapter 4 addresses scheduling in a stable state. This chapter addresses problems that occur when the system state changes. Indeed the C∀ runtime supports expanding and shrinking the number of processors, both manually and, to some extent, automatically. These changes affect the scheduling algorithm, which must dynamically alter its behaviour.

Specifically, C∀ supports adding processors using the type `processor`, in both RAII and heap coding scenarios.

```
{
	processor p[4]; // 4 new kernel threads
	... // execute on 4 processors
	processor * dp = new( processor, 6 ); // 6 new kernel threads
	... // execute on 10 processors
	delete( dp );   // delete 6 kernel threads
	... // execute on 4 processors
} // delete 4 kernel threads
```

Dynamically allocated processors can be deleted at any time, *i.e.*, their lifetime exceeds the block of creation. The consequence is that the scheduler and I/O subsystems must know when these processors come in and out of existence and roll them into the appropriate scheduling algorithms.

6.1 Manual Resizing

Manual resizing is expected to be a rare operation. Programmers normally create/delete processors on a cluster at startup/teardown. Therefore, dynamically changing the number of processors is an appropriate moment to allocate or free resources to match the new state. As such, all internal scheduling arrays that are sized based on the number of processors need to be `realloc`ed. This requirement also means any references into these arrays, *e.g.*, pointers or indexes, may need to be updated if elements are moved for compaction or any other reason.

There are no performance requirements, within reason, for act of resizing itself, since it is expected to be rare. However, this operation has strict correctness requirements, since updating and idle sleep can easily lead to deadlocks. The resizing mechanism should also avoid, as much as possible any effect on performance when the number of processors remains constant. This

last requirement prohibits naive solutions, like simply adding a global lock to the ready-queue arrays.

6.1.1 Read-Copy-Update

One solution is to use the Read-Copy-Update pattern [108]. This is a very common pattern that avoids large critical sections, which is why it is worth mentioning. In this pattern, resizing is done by creating a copy of the internal data structures, *e.g.*, see Figure 4.4, updating the copy with the desired changes, and then attempting an Indiana Jones Switch to replace the original with the copy. This approach has the advantage that it may not need any synchronization to do the switch, depending on how reclamation of the original copy is handled. However, there is a race where processors still use the original data structure after the copy is switched. This race not only requires adding a memory-reclamation scheme, but it also requires that operations made on the stale original version are eventually moved to the copy.

Specifically, the original data structure must be kept until all processors have witnessed the change. This requirement is the *memory reclamation challenge* and means every operation needs *some* form of synchronization. If all operations need synchronization, then the overall cost of this technique is likely to be similar to an uncontended lock approach. In addition to the classic challenge of memory reclamation, transferring the original data to the copy before reclaiming it poses additional challenges. Especially merging sub-queues while having a minimal impact on fairness and locality.

For example, given a linked list, having a node enqueued onto the original and new list is not necessarily a problem depending on the chosen list structure. If the list supports arbitrary insertions, then inconsistencies in the tail pointer do not break the list; however, ordering may not be preserved. Furthermore, nodes enqueued to the original queues eventually need to be uniquely transferred to the new queues, which may further perturb ordering. Dequeuing is more challenging when nodes appear on both lists because of pending reclamation: dequeuing a node from one list does not remove it from the other nor is that node in the same place on the other list. This situation can lead to multiple processors dequeuing the same thread. Fixing these challenges requires more synchronization or more indirection to the queues, plus coordinated searching to ensure unique elements.

6.1.2 Readers-Writer Lock

A simpler approach is to use a *Readers-Writer Lock* [102], where the resizing requires acquiring the lock as a writer while simply enqueueing/dequeuing threads requires acquiring the lock as a reader. Using a Readers-Writer lock solves the problem of dynamically resizing and leaves the challenge of finding or building a lock with sufficient good read-side performance. Since this approach is not a very complex challenge and an ad hoc solution is perfectly acceptable, building a Readers-Writer lock was the path taken.

To maximize reader scalability, readers should not contend with each other when attempting to acquire and release a critical section. Achieving this goal requires that each reader have its own memory to mark as locked and unlocked. The read-acquire possibly waits for a writer to finish

```
void read_lock() {
     // Step 1 : make sure no writers in
     while write_lock { Pause(); }
     // Step 2 : acquire our local lock
     while atomic_xchg( tls.lock ) { Pause(); }
}
void read_unlock() {
     tls.lock = false;
}
void write_lock()  {
     // Step 1 : lock global lock
     while atomic_xchg( write_lock ) { Pause(); }
     // Step 2 : lock per-proc locks
     for t in all_tls {
          while atomic_xchg( t.lock ) { Pause(); }
     }
}
void write_unlock() {
     // Step 1 : release local locks
     for t in all_tls { t.lock = false; }
     // Step 2 : release global lock
     write_lock = false;
}
```

Figure 6.1: Specialized Readers-Writer Lock

the critical section and then acquires a reader's local spinlock. The writer acquires the global lock, guaranteeing mutual exclusion among writers, and then acquires each of the local reader locks. Acquiring all the local read-locks guarantees mutual exclusion among the readers and the writer, while the wait on the read side prevents readers from continuously starving the writer. Figure 6.1 shows the outline for this specialized readers-writer lock. The lock is nonblocking, so both readers and writers spin while the lock is held. This very wide sharding strategy means that readers have very good locality since they only ever need to access two memory locations.

6.2 Idle-Sleep

While manual resizing of processors is expected to be rare, the number of threads can vary significantly over an application's lifetime, which means there are times when there are too few or too many processors. For this work, it is the application programmer's responsibility to manually create processors, so if there are too few processors, the application must address this issue. This leaves too many processors when there are not enough threads for all the processors to be useful. These idle processors cannot be removed because their lifetime is controlled by the application, and only the application knows when the number of threads may increase or decrease. While idle processors can spin until work appears, this approach wastes energy, unnecessarily produces heat and prevents other applications from using the hardware thread. Therefore, idle processors are put into an idle state, called *Idle-Sleep*, where the kernel-level thread is blocked until the

scheduler deems it is needed.

Idle sleep effectively encompasses several challenges. First, a data structure needs to keep track of all processors that are in idle sleep. Because idle sleep is spurious, this data structure has strict performance requirements, in addition to strict correctness requirements. Next, some mechanism is needed to block kernel-level threads, *e.g.*, `pthread_cond_wait` or a pthread semaphore. The complexity here is to support user-level locking, timers, I/O operations, and all other C∀ features with minimal complexity. Finally, the scheduler needs a heuristic to determine when to block and unblock an appropriate number of processors. However, this third challenge is outside the scope of this thesis because developing a general heuristic is complex enough to justify its own work. Therefore, the C∀ scheduler simply follows the "Race-to-Idle" [12] approach where a sleeping processor is woken any time a thread becomes ready and processors go to idle sleep anytime they run out of work.

An interesting subpart of this heuristic is what to do with bursts of threads that become ready. Since waking up a sleeping processor can have notable latency, multiple threads may become ready while a single processor is waking up. This fact raises the question: if many processors are available, how many should be woken? If the ready threads will run longer than the wake-up latency, waking one processor per thread will offer maximum parallelization. If the ready threads will run for a very short time, waking many processors may be wasteful. As mentioned, since a heuristic to handle these complex cases is outside the scope of this thesis, so the behaviour of the scheduler in this particular case is left unspecified.

6.3 Sleeping

As usual, the cornerstone of any feature related to the kernel is the choice of system call. In terms of blocking a kernel-level thread until some event occurs, the Linux kernel has many available options.

6.3.1 `pthread_mutex`/`pthread_cond`

The classic option is to use some combination of the pthread mutual exclusion and synchronization locks, allowing a safe park/unpark of a kernel-level thread to/from a `pthread_cond`. While this approach works for kernel-level threads waiting among themselves, I/O operations do not provide a mechanism to signal `pthread_cond`s. For I/O results to wake a processor waiting on a `pthread_cond` means a different kernel-level thread must be woken up first, which then signals the processor.

6.3.2 `io_uring` and Epoll

An alternative is to flip the problem on its head and block waiting for I/O, using `io_uring` or `epoll`. This creates the inverse situation, where I/O operations directly wake sleeping processors but waking blocked processors must use an indirect scheme. This generally takes the form of creating a file descriptor, *e.g.*, dummy file, pipe, or event fd, and using that file descriptor when processors need to wake each other. This leads to additional complexity because there can be a

race between these artificial I/O and genuine I/O operations. If not handled correctly, this can lead to artificial files getting delayed too long behind genuine files, resulting in longer latency.

6.3.3 Event FDs

Another interesting approach is to use an event file descriptor[35]. This Linux feature is a file descriptor that behaves like I/O, *i.e.*, uses `read` and `write`, but also behaves like a semaphore. Indeed, all reads and writes must use word-sized values, *i.e.*, 64 or 32 bits. Writes *add* their values to a buffer using arithmetic addition versus buffer append, and reads zero-out the buffer and return the buffer values so far.[1] If a read is made while the buffer is already 0, the read blocks until a non-0 value is added. What makes this feature particularly interesting is that `io_uring` supports the `IORING_REGISTER_EVENTFD` command to register an event `fd` to a particular instance. Once that instance is registered, any I/O completion results in `io_uring` writing to the event `fd`. This means that a processor waiting on the event `fd` can be *directly* woken up by either other processors or incoming I/O.

6.4 Tracking Sleepers

Tracking which processors are in idle sleep requires a data structure holding all the sleeping processors, but more importantly, it requires a concurrent *handshake* so that no thread is stranded on a ready queue with no active processor. The classic challenge occurs when a thread is made ready while a processor is going to sleep: there is a race where the new thread may not see the sleeping processor and the sleeping processor may not see the ready thread. Since threads can be made ready by timers, I/O operations, or other events outside a cluster, this race can occur even if the processor going to sleep is the only processor awake. As a result, improper handling of this race leads to all processors going to sleep when there are ready threads and the system deadlocks.

The handshake closing the race is done with both the notifier and the idle processor executing two ordered steps. The notifier first makes sure the newly ready thread is visible to processors searching for threads, and then attempts to notify an idle processor. On the other side, processors make themselves visible as idle processors and then search for any threads they may have missed. Unlike regular work-stealing, this search must be exhaustive to make sure that no pre-existing thread is missed. These steps from both sides guarantee that if the search misses a newly ready thread, then the notifier is guaranteed to see at least one idle processor. Conversely, if the notifier does not see any idle processor, then a processor is guaranteed to find the new thread in its exhaustive search.

Furthermore, the "Race-to-Idle" approach means that there may be contention on the data structure tracking sleepers. Contention can be tolerated for processors attempting to sleep or wake up because these processors are not doing useful work, and therefore, not contributing to overall performance. However, notifying, checking if a processor must be woken-up, and doing so if needed, can significantly affect overall performance and must be low cost.

[1] This behaviour is without the `EFD_SEMAPHORE` flag, which changes the behaviour of `read` but is not needed for this work.

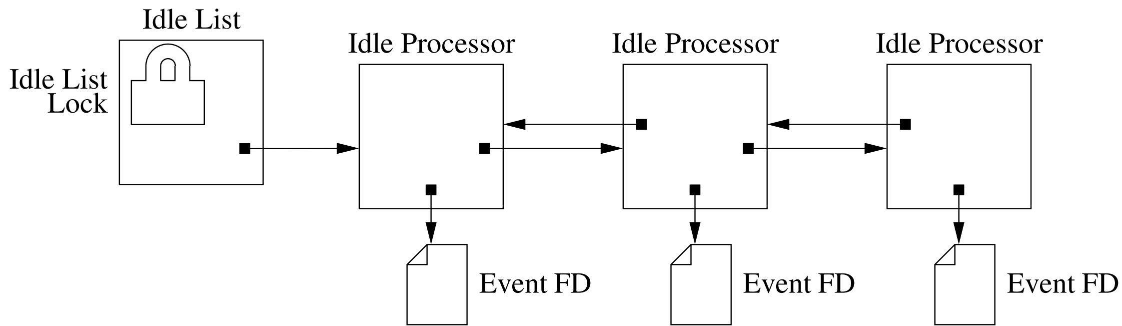


Figure 6.2: Basic Idle Sleep Data Structure
Each idle processor is put onto a doubly-linked stack protected by a lock. Each processor has a private event `fd`.

6.4.1 Sleepers List

Each cluster maintains a list of idle processors, organized as a stack. This ordering allows processors at the tail to stay in idle sleep for extended periods while those at the head of the list wake up for bursts of activity. Because of unbalanced performance requirements, the algorithm tracking sleepers is designed to have idle processors handle as much of the work as possible. The idle processors maintain the stack of sleepers among themselves and notifying a sleeping processor takes as little work as possible. This approach means that maintaining the list is fairly straightforward. The list can simply use a single lock per cluster and only processors that are getting in and out of the idle state contend for that lock.

This approach also simplifies notification. Indeed, processors not only need to be notified when a new thread is readied, but must also be notified during manual resizing, so the kernel-level thread can be joined. These requirements mean whichever entity removes idle processors from the sleeper list must be able to do so in any order. Using a simple lock over this data structure makes the removal much simpler than using a lock-free data structure. The single lock also means the notification process simply needs to wake up the desired idle processor, using `pthread_cond_signal`, `write` on an `fd`, *etc.*, and the processor handles the rest.

6.4.2 Reducing Latency

As mentioned in this section, processors going to sleep for extremely short periods is likely in certain scenarios. Therefore, the latency of doing a system call to read from and write to an event `fd` can negatively affect overall performance notably. Hence, it is important to reduce latency and contention of the notification as much as possible. Figure 6.2 shows the basic idle-sleep data structure. For the notifiers, this data structure can cause contention on the lock and the event `fd` syscall can cause notable latency.

Contention occurs because the idle-list lock must be held to access the idle list, *e.g.*, by processors attempting to go to sleep, processors waking, or notification attempts. The contention from the processors attempting to go to sleep can be mitigated slightly by using `try_acquire`, so the processors simply busy wait again searching for threads if the lock is held. This trick

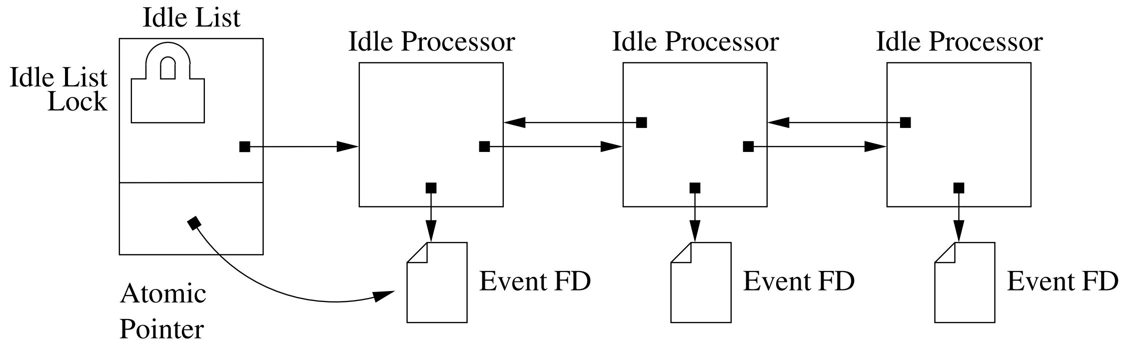


Figure 6.3: Improved Idle-Sleep Data Structure
An atomic pointer is added to the list pointing to the Event FD of the first processor on the list.

cannot be used when waking processors since the waker needs to return immediately to what it was doing.

Interestingly, general notification, *i.e.*, waking any idle processor versus a specific one, does not strictly require modifying the list. Here, contention can be reduced notably by having notifiers avoid the lock entirely by adding a pointer to the event `fd` of the first idle processor, as in Figure 6.3. To avoid contention among notifiers, notifiers atomically exchange the pointer with `NULL`. The first notifier succeeds on the exchange and obtains the `fd` of an idle processor; hence, only one notifier contends on the system call. This notifier writes to the `fd` to wake a processor. The woken processor then updates the atomic pointer, while it is updating the head of the list, as it removes itself from the list. Notifiers that obtained a `NULL` in the exchange simply move on knowing that another notifier is already waking a processor. This behaviour is equivalent to having multiple notifiers write to the `fd` since reads consume all previous writes. Note that with and without this atomic pointer, bursts of notification can lead to an unspecified number of processors being woken up, depending on how the arrival notification compares with the latency of processors waking up. As mentioned in section 6.2, there is no optimal approach to handle these bursts. It is therefore difficult to justify the cost of any extra synchronization here.

The next optimization is to avoid the latency of the event `fd`, which can be done by adding what is effectively a binary benaphore[76] in front of the event `fd`. The benaphore over the event `fd` logically provides a three-state flag to avoid unnecessary system calls, where the states are expressed explicitly in Figure 6.4. A processor begins its idle sleep by adding itself to the idle list before searching for a thread. In the process of adding itself to the idle list, it sets the state flag to `SEARCH`. If no threads can be found during the search, the processor then confirms it is going to sleep by atomically swapping the state to `SLEEP`. If the previous state is still `SEARCH`, then the processor does read the event `fd`. Meanwhile, notifiers atomically exchange the state to the `AWAKE` state. If the previous state is `SLEEP`, then the notifier must write to the event `fd`. However, if the notify arrives almost immediately after the processor marks itself idle, then both reads and writes on the event `fd` can be omitted, which reduces latency notably. These extensions lead to the final data structure shown in Figure 6.5.

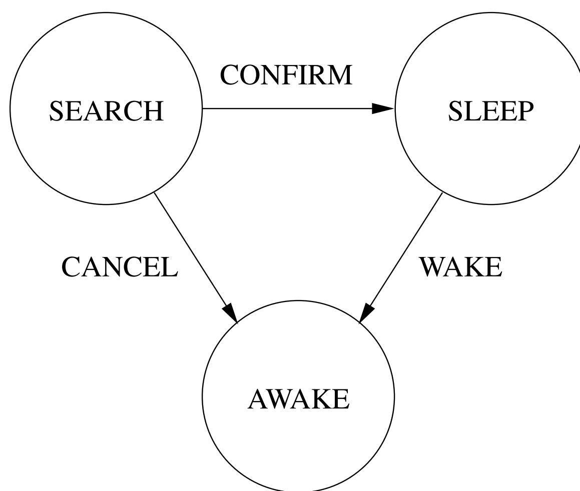


Figure 6.4: Improved Idle-Sleep Latency
A three-state flag is added to the event `fd`.

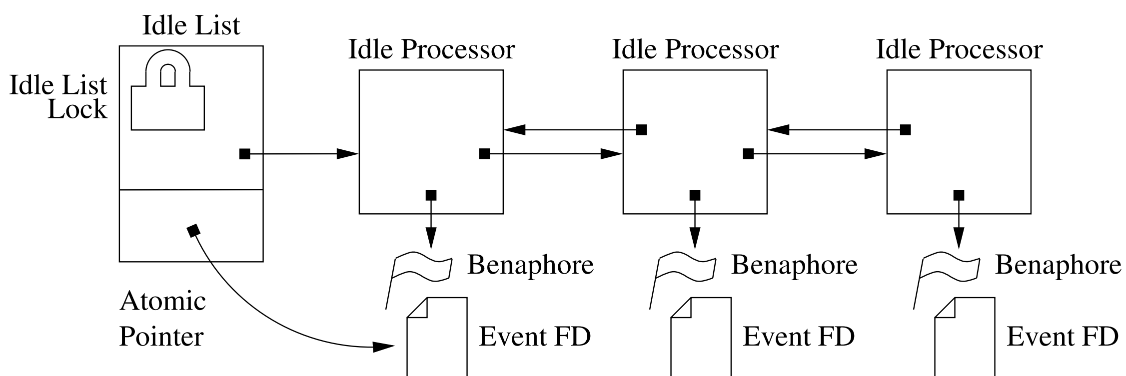


Figure 6.5: Low-latency Idle Sleep Data Structure
Each idle processor is put unto a doubly-linked stack protected by a lock. Each processor has a private event `fd` with a benaphore in front of it. The list also has an atomic pointer to the event `fd` and benaphore of the first processor on the list.

Part III

Evaluation

Chapter 7

Micro-Benchmarks

The first step in evaluating this work is to test small controlled cases to ensure the basics work properly. This chapter presents five different experimental setups for evaluating the basic features of the C∀, libfibre [48], Go, and Tokio [88] schedulers. All of these systems have a user-level threading model. The goal of this chapter is to show, through the different experiments, that the C∀ scheduler obtains equivalent performance to other schedulers with lesser fairness guarantees. Note that only the code of the C∀ tests is shown; all tests in the other systems are functionally identical and available both online [30] and submitted to UWSpace with the thesis itself.

7.1 Benchmark Environment

All benchmarks are run on two distinct hardware platforms.

AMD is a server with two AMD EPYC 7662 CPUs and 256GB of DDR4 RAM. The EPYC CPU has 64 cores with 2 hardware threads per core, for a total of 128 hardware threads per socket with 2 sockets for a total of 256 hardware threads. Each CPU has 4 MB, 64 MB and 512 MB of L1, L2 and L3 caches, respectively. Each L1 and L2 instance is only shared by hardware threads on a given core, but each L3 instance is shared by 4 cores, therefore 8 hardware threads. The server runs Ubuntu 20.04.2 LTS on top of Linux Kernel 5.8.0-55.

Intel is a server with four Intel Xeon Platinum 8160 CPUs and 384GB of DDR4 RAM. The Xeon CPU has 24 cores with 2 hardware threads per core, for 48 hardware threads per socket with 4 sockets for a total of 196 hardware threads. Each CPU has 3 MB, 96 MB and 132 MB of L1, L2 and L3 caches respectively. Each L1 and L2 instance are only shared by hardware threads on a given core, but each L3 instance is shared across the entire CPU, therefore 48 hardware threads. The server runs Ubuntu 20.04.2 LTS on top of Linux Kernel 5.8.0-55.

For all benchmarks, `taskset` is used to limit the experiment to 1 NUMA node with no hyperthreading. If more hardware threads are needed, then 1 NUMA node with hyperthreading is used. If still more hardware threads are needed, then the experiment is limited to as few NUMA nodes as needed. For the Intel machine, this means that from 1 to 24 processors one socket and *no* hyperthreading is used, and from 25 to 48 processors still only one socket is used but *with* hyperthreading. This pattern is repeated between 49 and 96, between 97 and 144, and between

145 and 192. On AMD, the same algorithm is used, but the machine only has 2 sockets. So hyperthreading[1] is used when the processor count reaches 65 and 193.

The limited sharing of the last-level cache on the AMD machine is markedly different from the Intel machine. Indeed, while on both architectures L2 cache misses that are served by L3 caches on a different CPU incur a significant latency, on the AMD it is also the case that cache misses served by a different L3 instance on the same CPU also incur high latency.

7.2 Experimental setup

Each experiment is run 15 times varying the number of processors depending on the two different computers. All experiments gather throughput data and secondary data for scalability or latency. The data is graphed using a solid, a dashed, and a dotted line, representing the median, maximum and minimum results respectively, where the minimum/maximum lines are referred to as the *extremes*.[2] This graph presentation offers an overview of the distribution of the results for each experiment.

For each experiment, four graphs are generated showing traditional throughput on the top row and *scalability* or *latency* on the bottom row (peek ahead to Figure 7.3). Scalability uses the same data as throughput but the Y-axis is calculated as the number of processors over the throughput. In this representation, perfect scalability should appear as a horizontal line, *e.g.*, if doubling the number of processors doubles the throughput, then the relation stays the same.

The left column shows results for hundreds of threads per processor, enough to always keep every processor busy. The right column shows results for very few threads per processor, where the ready queues are expected to be near empty most of the time. The distinction between many and few threads is meaningful because the idle sleep subsystem is expected to matter only in the right column, where spurious effects can cause a processor to run out of work temporarily.

7.3 Cycle

The most basic evaluation of any ready queue is the latency needed to push and pop one element from the ready queue. Since these two operations also describe a `yield` operation, many systems use this operation as the fundamental benchmark. However, yielding can be treated as a special case by optimizing it away since the number of ready threads does not change. Hence, systems that perform this optimization have an artificial performance benefit because the yield becomes a *nop*. For this reason, I designed a different push/pop benchmark, called *Cycle Benchmark*. This benchmark arranges several threads into a ring, as seen in Figure 7.1, where the ring is a circular singly-linked list. At runtime, each thread unparks the next thread before parking itself. Unparking the next thread pushes that thread onto the ready queue while the ensuing park leads to a thread being popped from the ready queue.

[1] Hyperthreading normally refers specifically to the technique used by Intel, however, it is often used generically to refer to any equivalent feature.

[2] An alternative display is to use error bars with min/max as the bottom/top for the bar. However, this approach is not truly an error bar around a mean value and I felt the connected lines are easier to read.

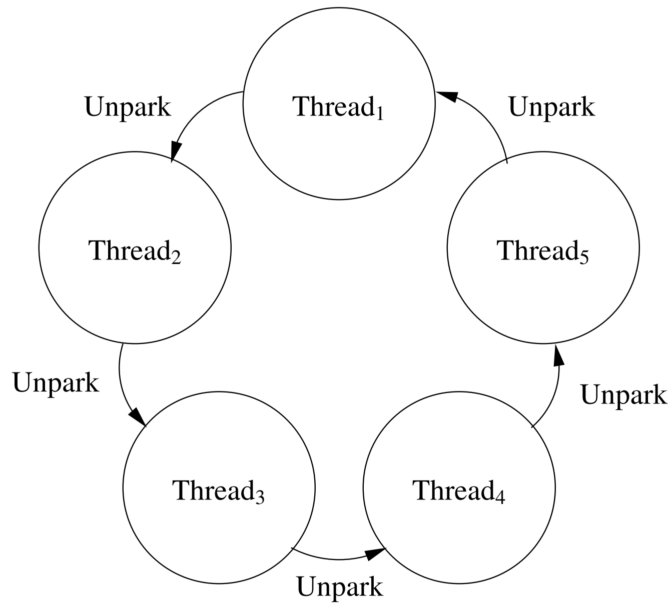


Figure 7.1: Cycle benchmark
Each thread unparks the next thread in the cycle before parking itself.

Therefore, the underlying runtime cannot rely on the number of ready threads staying constant over the duration of the experiment. In fact, the total number of threads waiting on the ready queue is expected to vary because of the race between the next thread unparking and the current thread parking. That is, the runtime cannot anticipate that the current task immediately parks. As well, the size of the cycle is also decided based on this race, *e.g.*, a small cycle may see the chain of unparks go full circle before the first thread parks because of time-slicing or multiple processors. If this happens, the scheduler push and pop are avoided and the results of the experiment are skewed. (Note, an unpark is like a V on a semaphore, so the subsequent park (P) may not block.) Every runtime system must handle this race and cannot optimize away the ready-queue pushes and pops. To prevent any attempt of silently omitting ready-queue operations, the ring of threads is made big enough so the threads have time to fully park before being unparked again. Finally, to further mitigate any underlying push/pop optimizations, especially on SMP machines, multiple rings are created in the experiment.

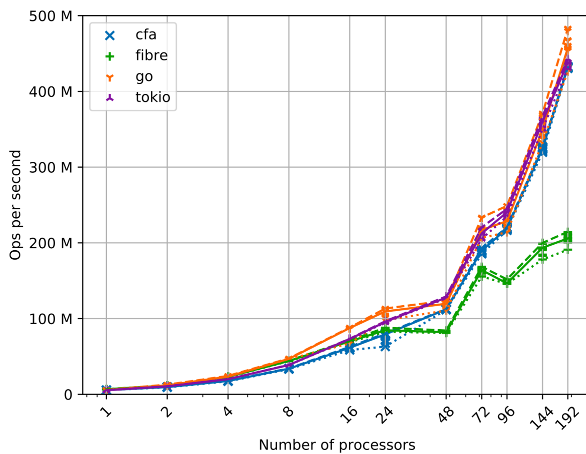
Figure 7.2 shows the pseudo code for this benchmark, where each cycle has 5 threads. There is additional complexity to handle termination (not shown), which requires a binary semaphore or a channel instead of raw park/unpark and carefully picking the order of the `P` and `V` with respect to the loop condition.

7.3.1 Results

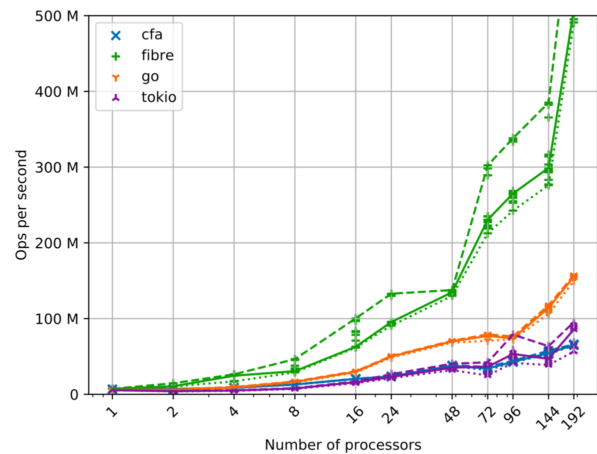
Figures 7.3 and 7.4 show the results for the cycle experiment on Intel and AMD, respectively. Looking at the left column on Intel, Figures 7.3a and 7.3c show the results for 100 cycles of 5 threads for each processor. C∀, Go and Tokio all obtain effectively the same throughput performance. Libfibre is slightly behind in this case but still scales decently. As a result of

```
Thread.main() {
     count := 0
     for {
          this.next.wake()
          wait()
          count ++
          if must_stop() { break }
     }
     global.count += count
}
```

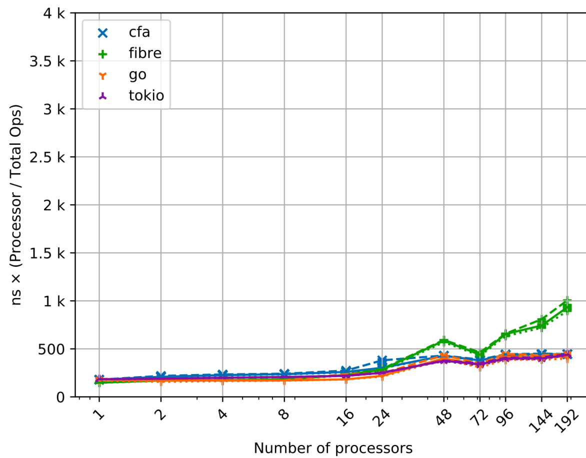
Figure 7.2: Cycle Benchmark: Pseudo Code



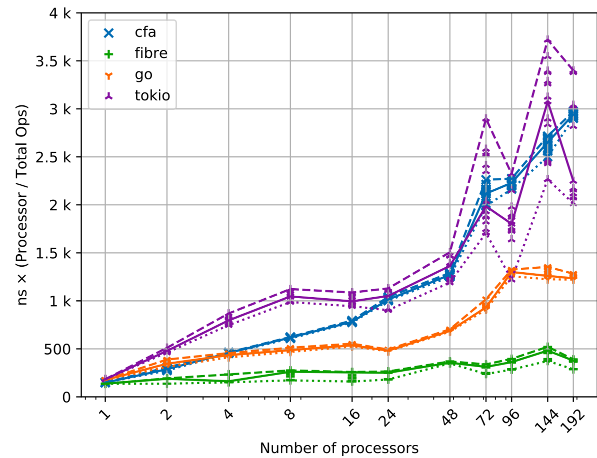
(a) Throughput, 100 cycles per processor



(b) Throughput, 1 cycle per processor

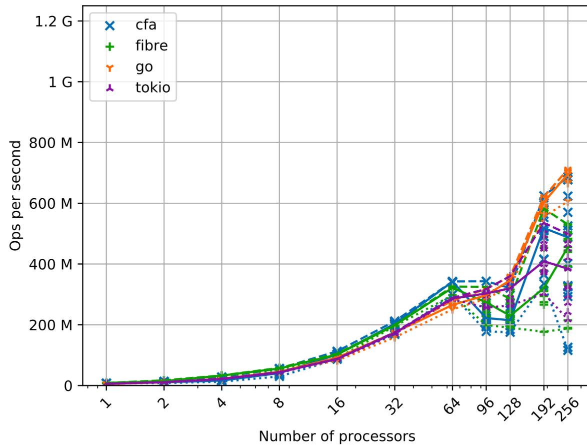


(c) Scalability, 100 cycles per processor

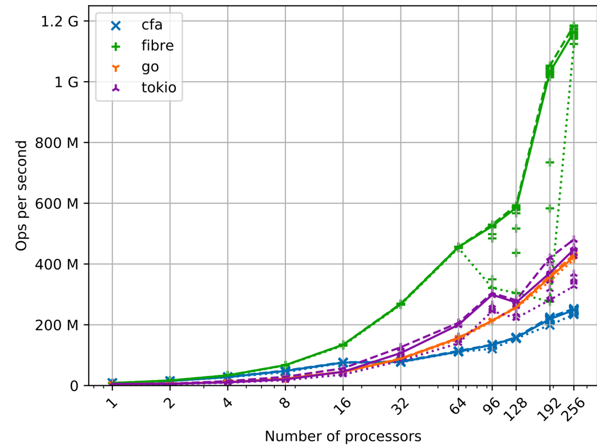


(d) Scalability, 1 cycle per processor

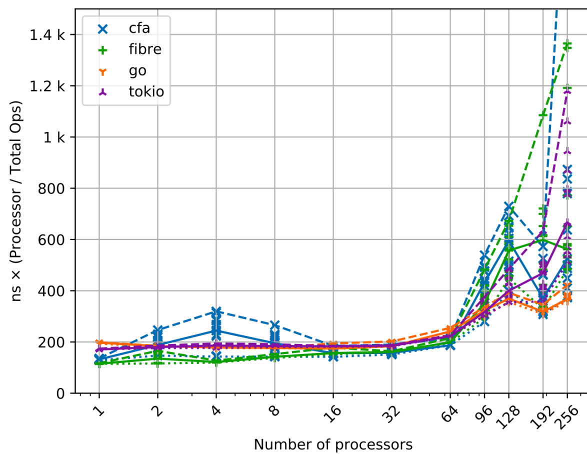
Figure 7.3: Cycle Benchmark on Intel
Throughput and scalability as a function of processor count, 5 threads per cycle, and different cycle counts. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.



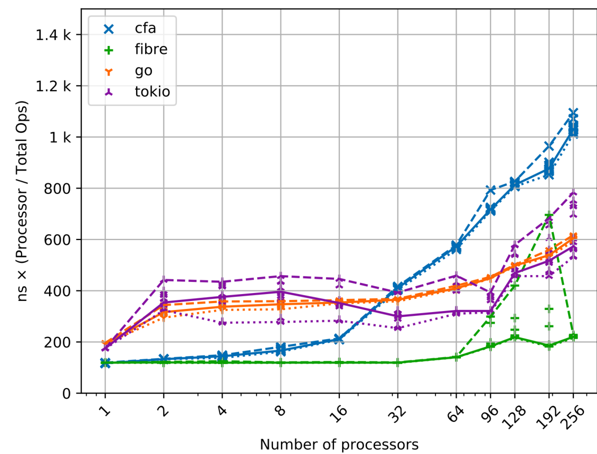
(a) Throughput, 100 cycles per processor



(b) Throughput, 1 cycle per processor



(c) Scalability, 100 cycles per processor



(d) Scalability, 1 cycle per processor

Figure 7.4: Cycle Benchmark on AMD
Throughput and scalability as a function of processor count, 5 threads per cycle, and different cycle counts. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.

the kernel-level thread placement, additional processors from 25 to 48 offer less performance improvement for all runtimes, which can be seen as a flattening of the line. This effect even causes a decrease in throughput in libfibre's case. As expected, this pattern repeats between processor count 72 and 96.

Looking next at the right column on Intel, Figures 7.3b and 7.3d show the results for 1 cycle of 5 threads for each processor. C∀ and Tokio obtain very similar results overall, but Tokio shows more variations in the results. Go achieves slightly better performance than C∀ and Tokio, but all three display significantly worse performance compared to the left column. This decrease in performance is likely due to the additional overhead of the idle-sleep mechanism. This can either be the result of processors actually running out of work or simply additional overhead

from tracking whether or not there is work available. Indeed, unlike the left column, it is likely that the ready queue is transiently empty, which likely triggers additional synchronization steps. Interestingly, libfibre achieves better performance with 1 cycle.

Looking now at the results for the AMD architecture, Figure 7.4, the results are overall similar to the Intel results, but with close to double the performance, slightly increased variation, and some differences in the details. Note the maximum of the Y-axis on Intel and AMD differ significantly. Looking at the left column on AMD, Figures 7.4a and 7.4c, all 4 runtimes achieve very similar throughput and scalability. However, as the number of processors grows higher, the results on AMD show notably more variability than on Intel. The different performance improvements and plateaus are due to cache topology and appear at the expected processor counts of 64, 128 and 192, for the same reasons as on Intel. Looking next at the right column on AMD, Figures 7.4b and 7.4d, Tokio and Go have the same throughput performance, while C∀ is slightly slower. This result is different than on Intel, where Tokio behaved like C∀ rather than behaving like Go. Again, the same performance increase for libfibre is visible when running fewer threads. I did not investigate the libfibre performance boost for 1 cycle in this experiment.

The conclusion from both architectures is that all of the compared runtimes have fairly equivalent performance for this micro-benchmark. Clearly, the pathological case with 1 cycle per processor can affect fairness algorithms managing mostly idle processors, *e.g.*, C∀, but only at high core counts. In this case, *any* helping is likely to cause a cascade of processors running out of work and attempting to steal. For this experiment, the C∀ scheduler has achieved the goal of obtaining equivalent performance to other schedulers with lesser fairness guarantees.

7.4 Yield

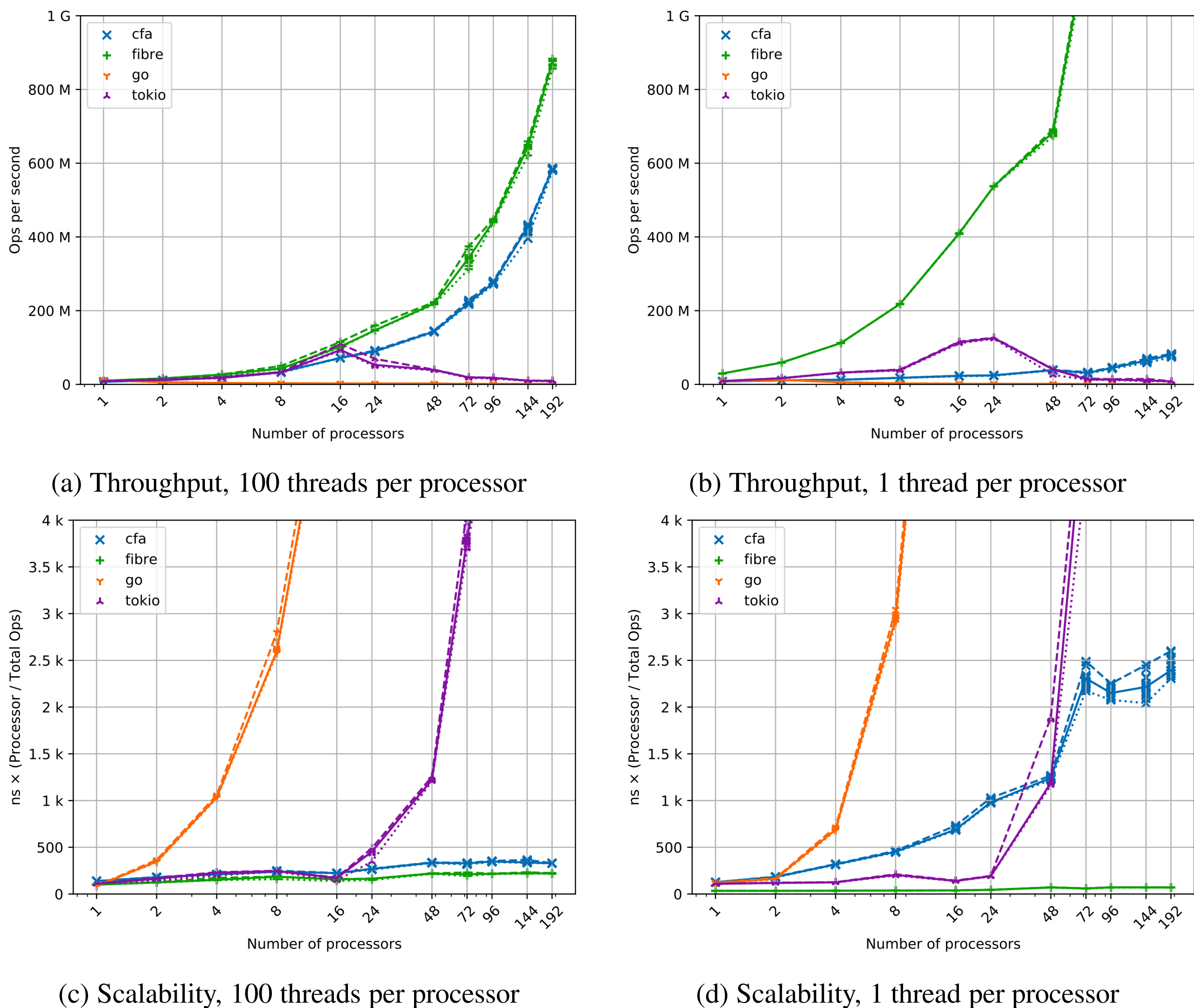
For completeness, the classic yield benchmark is included. Here, the throughput is dominated by the mechanism used to handle the `yield` function. Figure 7.5 shows pseudo code for this benchmark, where the cycle `wait/next.wake` is replaced by `yield`. As mentioned, this benchmark may not be representative because of optimization shortcuts in `yield`. The only interesting variable in this benchmark is the number of threads per processors, where ratios close to 1 means the ready queue(s) can be empty, which again puts a strain on the idle-sleep handling.

7.4.1 Results

Figures 7.6 and 7.7 show the results for the yield experiment on Intel and AMD, respectively. Looking at the left column on Intel, Figures 7.6a and 7.6c show the results for 100 threads for each processor. Note that the Y-axis on this graph is twice as large as the Intel cycle graph. A visual glance between the left columns of the cycle and yield graphs confirms my claim that the yield benchmark is unreliable. C∀ has no special handling for `yield`, but this experiment requires less synchronization than the `cycle` experiment. Hence, the `yield` throughput and scalability graphs have similar shapes to the corresponding `cycle` graphs. The only difference is slightly better performance for `yield` because of less synchronization. Libfibre has special handling for `yield` using the fact that the number of ready fibres does not change, and therefore, bypassing the idle-sleep mechanism entirely. Hence, libfibre behaves very differently in the cycle

```
Thread.main() {
     count := 0
     for {
          yield()
          count ++
          if must_stop() { break }
     }
     global.count += count
}
```

Figure 7.5: Yield Benchmark: Pseudo Code



(a) Throughput, 100 threads per processor

(b) Throughput, 1 thread per processor

(c) Scalability, 100 threads per processor

(d) Scalability, 1 thread per processor

Figure 7.6: Yield Benchmark on Intel

Throughput and scalability as a function of processor count. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.

and yield benchmarks, with a 4 times increase in performance on the left column. Go has special handling for `yield` by putting a yielding goroutine on a secondary global ready-queue, giving it a lower priority. The result is that multiple hardware threads contend for the global queue and performance suffers drastically. Hence, Go behaves very differently in the cycle and yield benchmarks, with a complete performance collapse in `yield`. Tokio has a similar performance collapse after 16 processors, and therefore, its special `yield` handling is probably related to a Go-like scheduler problem and/or a C∀ idle-sleep problem. (I did not dig through the Rust code to ascertain the exact reason for the collapse.) Note that since there is no communication among threads, locality problems are much less likely than for the cycle benchmark. This lack of communication is probably why the plateaus due to topology are not present.

Looking next at the right column on Intel, Figures 7.6b and 7.6d show the results for 1 thread for each processor. As for `cycle`, C∀'s cost of idle sleep comes into play in a very significant way in Figure 7.6d, where the scaling is not flat. This result is to be expected since fewer threads mean processors are more likely to run out of work. On the other hand, when only running 1 thread per processor, libfibre optimizes further and forgoes the context switch entirely. This results in libfibre outperforming other runtimes, even more, achieving 8 times more throughput than for `cycle`. Finally, Go and Tokio's performance collapse is still the same with fewer threads. The only exception is Tokio running on 24 processors, deepening the mystery of its yielding mechanism further.

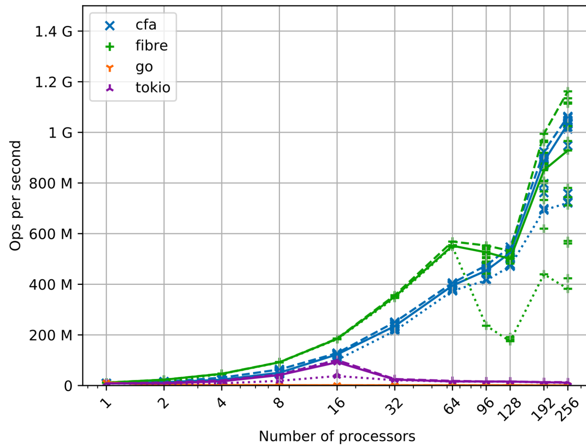
Looking now at the results for the AMD architecture, Figure 7.7, the results again show a story that is overall similar to the results on the Intel, with increased variation and some differences in the details. Note that the maximum of the Y-axis on Intel and AMD differ less in `yield` than `cycle`. Looking at the left column first, Figures 7.7a and 7.7c, C∀ achieves very similar throughput and scaling. Libfibre still outpaces all other runtimes, but it encounters a performance hit at 64 processors. This anomaly suggests some amount of communication between the processors that the Intel machine is able to mask where the AMD is not, once hyperthreading is needed. Go and Tokio still display the same performance collapse as on Intel. Looking next at the right column on AMD, Figures 7.7b and 7.7d, all runtime systems effectively behave the same as they did on the Intel machine. At the high threads count, the only difference is Libfibre's scaling and this difference disappears on the right column. This behaviour suggests whatever communication issue it encountered on the left is completely circumvented on the right.

It is difficult to draw conclusions for this benchmark when runtime systems treat `yield` so differently. The win for C∀ is its consistency between the cycle and yield benchmarks, making it simpler for programmers to use and understand, *i.e.*, the C∀ semantics match with programmer intuition.

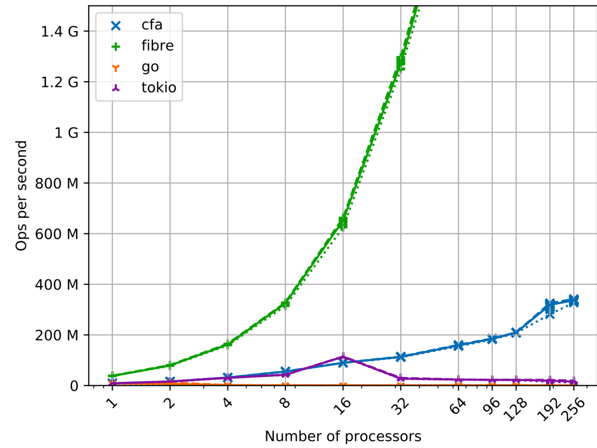
7.5 Churn

The Cycle and Yield benchmarks represent an *easy* scenario for a scheduler, *e.g.*, an embarrassingly parallel application. In these benchmarks threads can be easily partitioned over the different processors upfront and none of the threads communicate with each other.

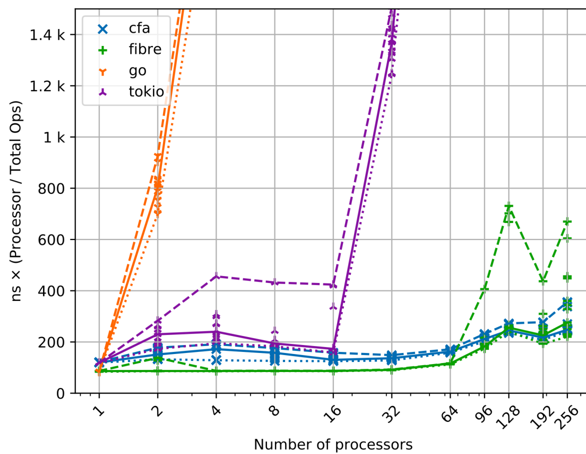
The Churn benchmark represents more chaotic executions, where there is more communication among threads but no relationship between the last processor on which a thread ran and



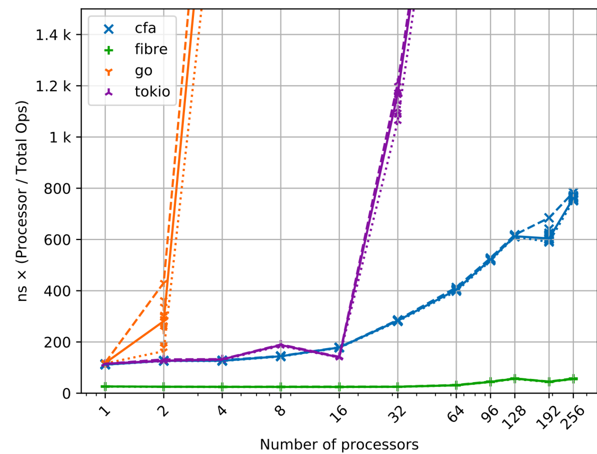
(a) Throughput, 100 threads per processor



(b) Throughput, 1 thread per processor



(c) Scalability, 100 threads per processor



(d) Scalability, 1 thread per processor

Figure 7.7: Yield Benchmark on AMD

Throughput and scalability as a function of processor count. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.

blocked, and the processor that subsequently unblocks it. With processor-specific ready-queues, when a thread is unblocked by a different processor, that means the unblocking processor must either "steal" the thread from another processor or find it on a remote queue. This dequeuing results in either contention on the remote queue and/or remote memory references on the thread data structure. Hence, this benchmark has performance dominated by the cache traffic as processors are constantly accessing each others' data. In either case, this benchmark aims to measure how well a scheduler handles these cases since both cases can lead to performance degradation if not handled correctly.

This benchmark uses a fixed-size array of counting semaphores. Each thread picks a random semaphore, `V`s it to unblock any waiting thread, and then `P`s (maybe blocks) the thread on the

semaphore. This creates a flow where threads push each other out of the semaphores before being pushed out themselves. For this benchmark to work, the number of threads must be equal to or greater than the number of semaphores plus the number of processors; *e.g.*, if there are 10 semaphores and 5 processors, but only 3 threads, all 3 threads can block (P) on a random semaphore and now there are no threads to unblock (V) them. Note that the nature of these semaphores means the counter can go beyond 1, which can lead to nonblocking calls to `P`. Figure 7.8 shows pseudo code for this benchmark, where the `yield` is replaced by `V` and `P`.

7.5.1 Results

Figures 7.9 and Figure 7.10 show the results for the churn experiment on Intel and AMD, respectively. Looking at the left column on Intel, Figures 7.9a and 7.9c show the results for 100 threads for each processor, and all runtimes obtain fairly similar throughput for most processor counts. C∀ does very well on a single processor but quickly loses its advantage over the other runtimes. As expected, it scales decently up to 48 processors, drops from 48 to 72 processors, and then plateaus. Tokio achieves very similar performance to C∀, with the starting boost, scaling decently until 48 processors, drops from 48 to 72 processors, and starts increasing again to 192 processors. Libfibre obtains effectively the same results as Tokio with slightly less scaling, *i.e.*, the scaling curve is the same but with slightly lower values. Finally, Go gets the most peculiar results, scaling worse than other runtimes until 48 processors. At 72 processors, the results of the Go runtime vary significantly, sometimes scaling sometimes plateauing. However, beyond this point Go keeps this level of variation but does not scale further in any of the runs.

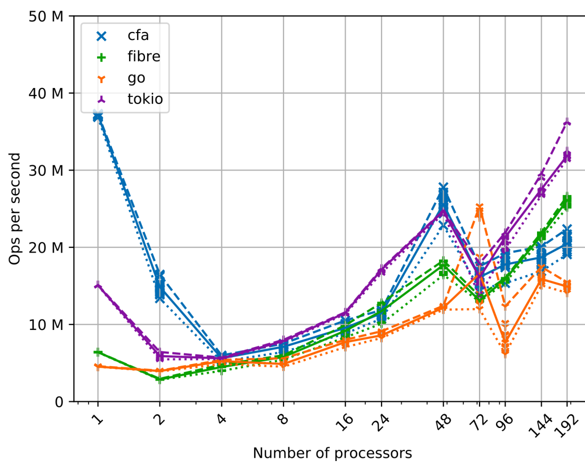
Throughput and scalability are notably worse for all runtimes than the previous benchmarks since there is inherently more communication between processors. Indeed, none of the runtimes reach 40 million operations per second while in the cycle benchmark all but libfibre reached 400 million operations per second. Figures 7.9c and 7.9d show that for all processor counts, all runtimes produce poor scaling. However, once the number of hardware threads goes beyond a single socket, at 48 processors, scaling goes from bad to worse and performance completely ceases to improve. At this point, the benchmark is dominated by inter-socket communication costs for all runtimes.

An interesting aspect to note here is that the runtimes differ in how they handle this situation. Indeed, when a processor unparks a thread that was last run on a different processor, the thread could be appended to the ready queue of the local processor or to the ready queue of the remote processor, which previously ran the thread. C∀, Tokio and Go all use the approach of unparking to the local processor, while Libfibre unparks to the remote processor. In this particular benchmark, the inherent chaos of the benchmark, in addition to the small memory footprint, means neither approach wins over the other.

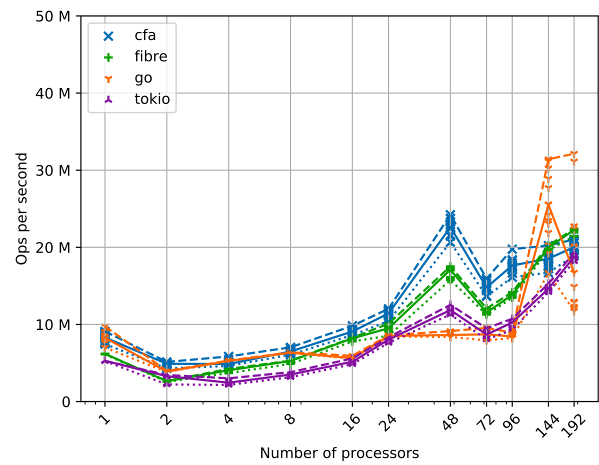
Looking next at the right column on Intel, Figures 7.9b and 7.9d show the results for 1 thread for each processor, and many of the differences between the runtimes disappear. C∀ outperforms other runtimes by a minuscule margin. Libfibre follows very closely behind with basically the same performance and scaling. Tokio maintains effectively the same curve shapes as C∀ and libfibre, but it incurs extra costs for all processor counts. While Go maintains overall similar results to the others, it again encounters significant variation at high processor counts.

```
Thread.main() {
    count := 0
    for {
        r := random() % len(spots)
        spots[r].V()
        spots[r].P()
        count ++
        if must_stop() { break }
    }
    global.count += count
}
```

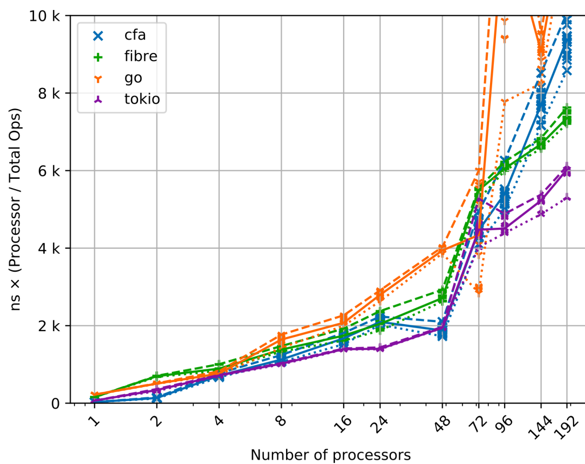
Figure 7.8: Churn Benchmark: Pseudo Code



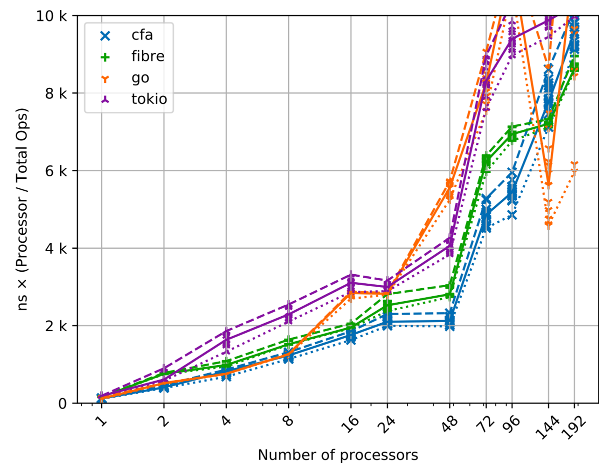
(a) Throughput, 100 threads per processor



(b) Throughput, 2 threads per processor



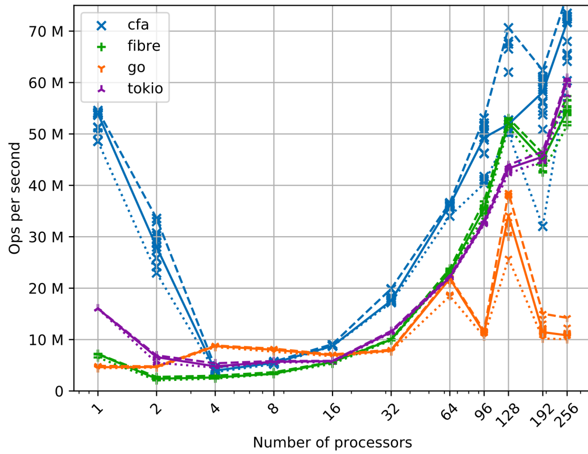
(c) Scalability, 100 threads per processor



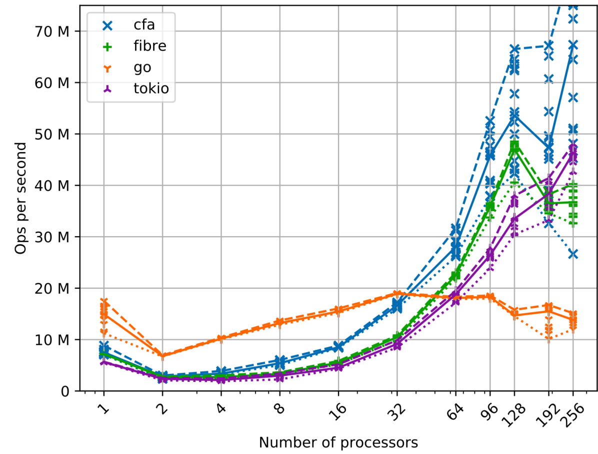
(d) Scalability, 2 threads per processor

Figure 7.9: Churn Benchmark on Intel

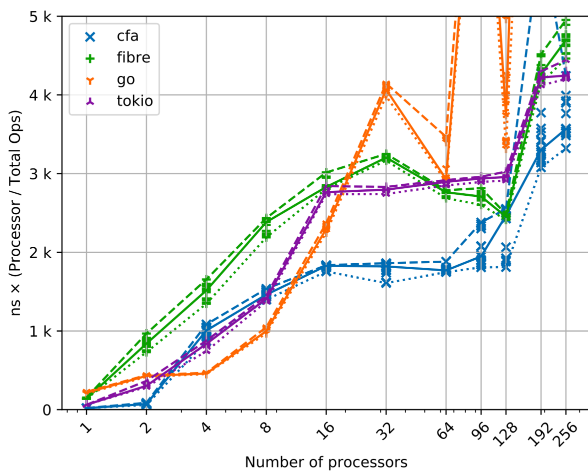
Throughput and scalability as a function of processor count. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.



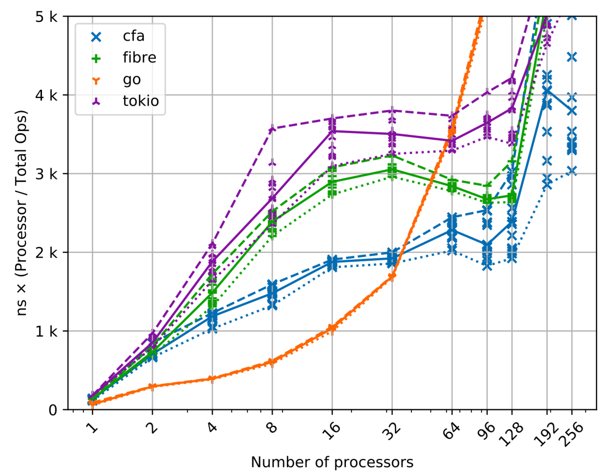
(a) Throughput, 100 threads per processor



(b) Throughput, 2 threads per processor



(c) Scalability, 100 threads per processor



(d) Scalability, 2 threads per processor

Figure 7.10: Churn Benchmark on AMD

Throughput and scalability as a function of processor count. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.

Inexplicably resulting in super-linear scaling for some runs, *i.e.*, the scalability curves display a negative slope.

Interestingly, unlike the cycle benchmark, running with fewer threads does not produce drastically different results. In fact, the overall throughput stays almost exactly the same on the left and right columns.

Looking now at the results for the AMD architecture, Figure 7.10, the results show a somewhat different story. Looking at the left column first, Figures 7.10a and 7.10c, C∀, Libfibre and Tokio all produce decent scalability. C∀ suffers particularly from larger variations at higher processor counts, but largely outperforms the other runtimes. Go still produces intriguing results

in this case and even more intriguingly, the results have fairly low variation.

One possible explanation for Go's difference is that it has very few available concurrent primitives, so a channel is substituted for a semaphore. On paper, a semaphore can be replaced by a channel, and with zero-sized objects passed through the channel, equivalent performance could be expected. However, in practice, there are implementation differences between the two, *e.g.*, if the semaphore count can get somewhat high so objects accumulate in the channel. Note that this substitution is also made in the cycle benchmark; however, in that context, it did not have a notable impact.

A second possible explanation is that Go may use the heap when allocating variables based on the result of the escape analysis of the code. It is possible for variables that could be placed on the stack to instead be placed on the heap. This placement could cause extra pointer chasing in the benchmark, heightening locality effects. Depending on how the heap is structured, this could also lead to false sharing. I did not investigate what causes these unusual results.

Looking next at the right column, Figures 7.10b and 7.10d, as for Intel, all runtimes obtain overall similar throughput between the left and right column. C∀, Libfibre and Tokio all have very close results. Go still suffers from poor scalability but is now unusual in a different way. While it obtains effectively constant performance regardless of processor count, this "sequential" performance is higher than the other runtimes for low processor count. Up to 32 processors, after which the other runtimes manage to outscale Go.

In conclusion, the objective of this benchmark is to demonstrate that unparking threads from remote processors does not cause too much contention on the local queues. Indeed, the fact that most runtimes achieve some scaling between various processor counts demonstrates migrations do not need to be serialized. Again these results demonstrate that C∀ achieves satisfactory performance compared to the other runtimes.

7.6 Locality

As mentioned in the churn benchmark, when unparking a thread, it is possible to either unpark to the local or remote sub-queue.[3] The locality experiment includes two variations of the churn benchmark, where a data array is added. In both variations, before `V`ing the semaphore, each thread calls a `work` function which increments random cells inside the data array. In the noshare variation, the array is not passed on and each thread continuously accesses its private array. In the share variation, the array is passed to another thread via the semaphore's shadow queue (each blocking thread can save a word of user data in its blocking node), transferring ownership of the array to the woken thread. Figure 7.11 shows the pseudo code for this benchmark.

The objective here is to highlight the different decisions made by the runtime when unparking. Since each thread unparks a random semaphore, it means that it is unlikely that a thread is unparked from the last processor it ran on. In the noshare variation, unparking the thread on the local processor is an appropriate choice since the data was last modified on that processor. In the shared variation, unparking the thread on a remote processor is an appropriate choice.

[3]It is also possible to unpark to a third unrelated sub-queue, but without additional knowledge about the situation, it is likely to degrade performance.

```
Thread.main() {
   count := 0
   for {
      r := random() % len(spots)
      // go through the array
      work( a )

      spots[r].V()
      spots[r].P()
      count ++
      if must_stop() { break }
   }
   global.count += count
}
```

(a) Noshare

```
Thread.main() {
   count := 0
   for {
      r := random() % len(spots)
      // go through the array
      work( a )
      // pass array to next thread
      spots[r].V( a )
      a = spots[r].P()
      count ++
      if must_stop() { break }
   }
   global.count += count
}
```

(b) Share

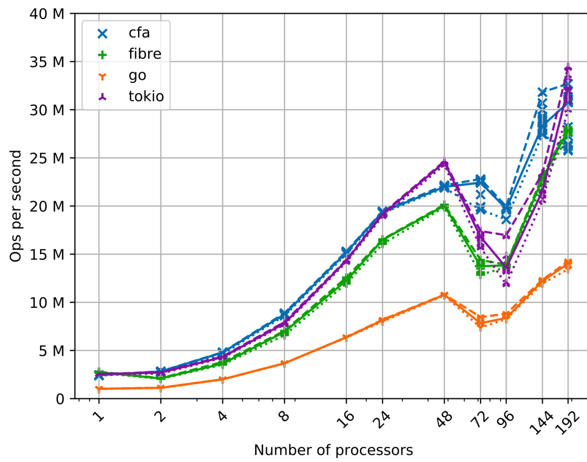
Figure 7.11: Locality Benchmark: Pseudo Code

The expectation for this benchmark is to see a performance inversion, where runtimes fare notably better in the variation which matches their unparking policy. This decision should lead to C∀, Go and Tokio achieving better performance in the share variation while libfibre achieves better performance in noshare. Indeed, C∀, Go and Tokio have the default policy of unparking threads on the local processor, whereas libfibre has the default policy of unparking threads wherever they last ran.

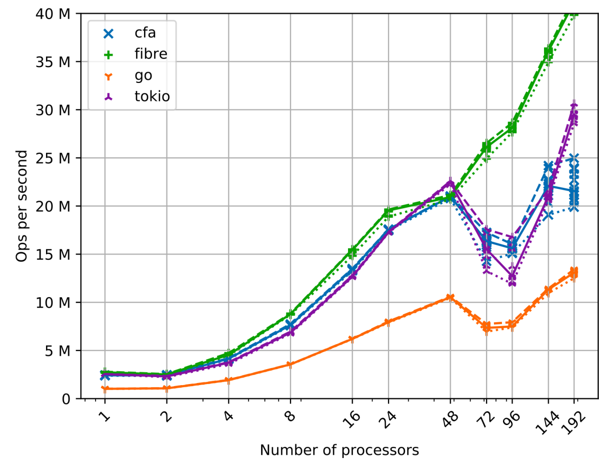
7.6.1 Results

Figures 7.12 and 7.13 show the results for the locality experiment on Intel and AMD, respectively. In both cases, the graphs on the left column show the results for the share variation and the graphs on the right column show the results for the noshare. Looking at the left column on Intel, Figures 7.12a and 7.12c show the results for the share variation. C∀ and Tokio slightly outperform libfibre, as expected, based on their threads placement approach. C∀ and Tokio both unpark locally and do not suffer cache misses on the transferred array. Libfibre, on the other hand, unparks remotely, and as such the unparked thread is likely to miss on the shared data. Go trails behind in this experiment, presumably for the same reasons that were observable in the churn benchmark. Otherwise, the results are similar to the churn benchmark, with lower throughput due to the array processing. As for most previous results, all runtimes suffer a performance hit after 48 processors, which is the socket boundary, and climb again from 96 to 192 processors.

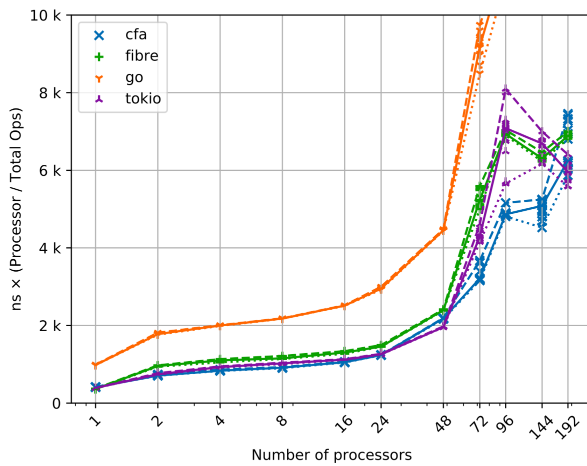
Looking at the right column on Intel, Figures 7.12b and 7.12d show the results for the noshare variation. The graphs show the expected performance inversion where libfibre now outperforms C∀ and Tokio. Indeed, in this case, unparking remotely means the unparked thread is less likely to suffer a cache miss on the array, which leaves the thread data structure and the remote queue as the only source of likely cache misses. Results show both are amortized fairly well in this case. C∀ and Tokio both unpark locally and as a result, suffer a marginal performance degradation



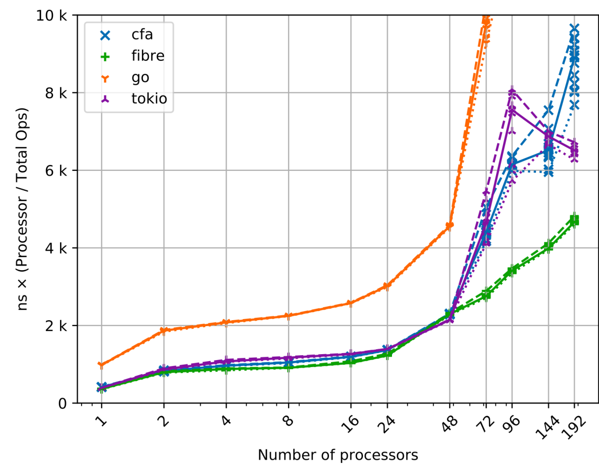
(a) Throughput share



(b) Throughput noshare



(c) Scalability share



(d) Scalability noshare

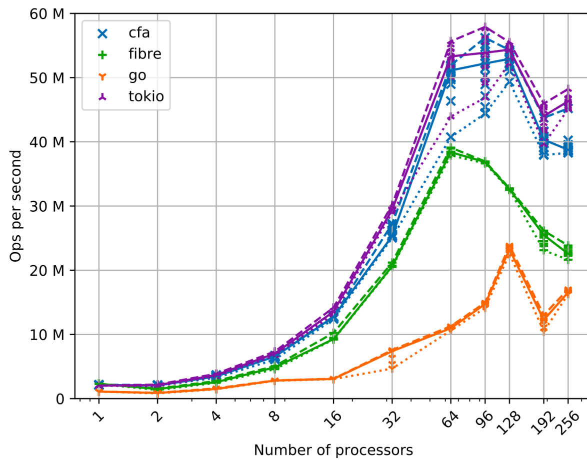
Figure 7.12: Locality Benchmark on Intel

Throughput and scalability as a function of processor count. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.

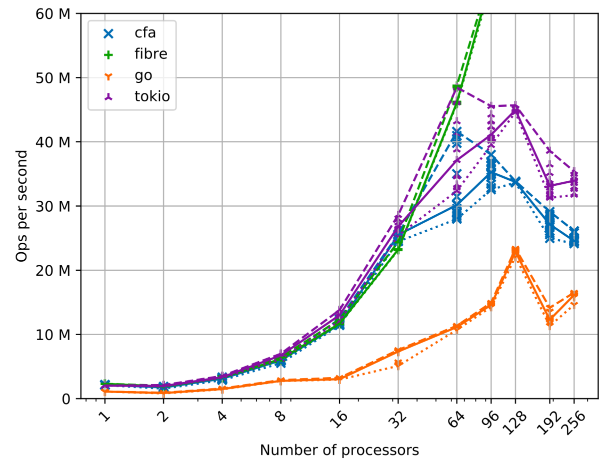
from the cache miss on the array.

Looking at the results for the AMD architecture, Figure 7.13, shows results similar to the Intel. Again the overall performance is higher and slightly more variation is visible. Looking at the left column first, Figures 7.13a and 7.13c, C∀ and Tokio still outperform libfibre, this time more significantly. This advantage is expected from the AMD server with its smaller and narrower caches that magnify the costs of processing the array. Go still has the same poor performance as on Intel.

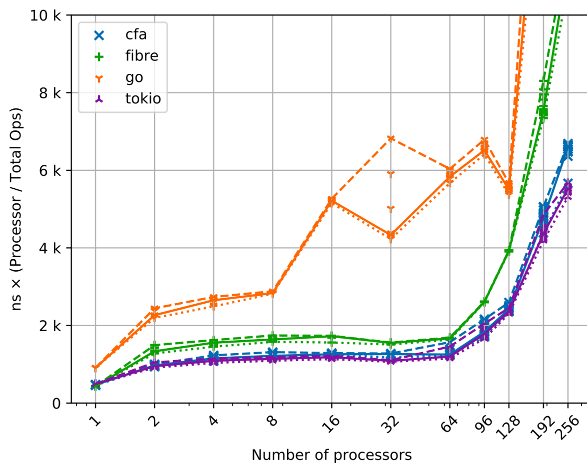
Finally, looking at the right column, Figures 7.13b and 7.13d, like on Intel, the same performance inversion is present between libfibre and C∀/Tokio. Go still has the same poor perfor-



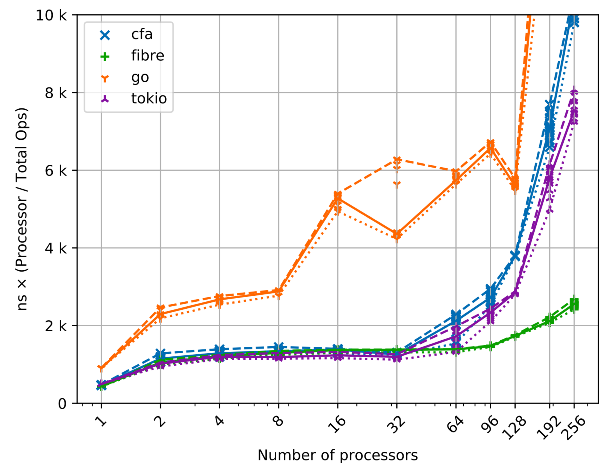
(a) Throughput share



(b) Throughput noshare



(c) Scalability share



(d) Scalability noshare

Figure 7.13: Locality Benchmark on AMD

Throughput and scalability as a function of processor count. For throughput, higher is better, for scalability, lower is better. Each series represents 15 independent runs. The dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.

mance.

Overall, this benchmark mostly demonstrates the two options available when unparking a thread. Depending on the workload, either of these options can be the appropriate one. Since it is prohibitively difficult to dynamically detect which approach is appropriate, all runtimes much choose one of the two and live with the consequences.

Once again, these experiments demonstrate that C∀ achieves equivalent performance to the other runtimes, in this case matching the faster Tokio rather than Go, which is trailing behind.

```
Thread.lead() {
     this.idx_seen = ++lead_idx
     if lead_idx > stop_idx {
          done := true
          return
     }
     // Wait for everyone to acknowledge my leadership
     start: = timeNow()
     for t in threads {
          while t.idx_seen != lead_idx {
               asm pause
               if (timeNow() - start) > 5 seconds { error() }
          }
     }
     // pick next leader
     leader := threads[ prng() % len(threads) ]
     // wake everyone
     if ! exhaust {
          for t in threads {
               if t != me { t.wake() }
          }
     }
}
Thread.wait() {
     this.idx_seen := lead_idx
     if exhaust { wait() }
     else { yield() }
}
Thread.main() {
     while !done  {
          if leader == me { this.lead() }
          else { this.wait() }
     }
}
```

Figure 7.14: Transfer Benchmark: Pseudo Code

7.7 Transfer

The last benchmark is more of an experiment than a benchmark. It tests the behaviour of the schedulers for a misbehaved workload. In this workload, one thread is selected at random to be the leader. The leader then spins in a tight loop until it has observed that all other threads have acknowledged its leadership. The leader thread then picks a new thread to be the next leader and the cycle repeats. The benchmark comes in two variations for the non-leader threads: once they acknowledged the leader, they either block on a semaphore or spin yielding. Figure 7.14 shows pseudo code for this benchmark.

The experiment is designed to evaluate the short-term load balancing of a scheduler. Indeed,

schedulers where the runnable threads are partitioned on the processors may need to balance the threads for this experiment to terminate. This problem occurs because the spinning thread is effectively preventing the processor from running any other thread. In the semaphore variation, the number of runnable threads eventually dwindles to only the leader. This scenario is a simpler case to handle for schedulers since processors eventually run out of work. In the yielding variation, the number of runnable threads stays constant. This scenario is a harder case to handle because corrective measures must be taken even when work is available. Note that runtimes with preemption circumvent this problem by forcing the spinner to yield. In C∀ preemption was disabled as it only obfuscates the results. I am not aware of a method to disable preemption in Go.

In both variations, the experiment effectively measures how long it takes for all threads to run once after a given synchronization point. In an ideal scenario where the scheduler is strictly FIFO, every thread would run once after the synchronization and therefore the delay between leaders would be given by, $NT(CSL+SL)/(NP-1)$, where CSL is the context-switch latency, SL is the cost for enqueueing and dequeuing a thread, NT is the number of threads, and NP is the number of processors. However, if the scheduler allows threads to run many times before other threads can run once, this delay increases. The semaphore version is an approximation of strictly FIFO scheduling, where none of the threads *attempt* to run more than once. The benchmark effectively provides the fairness guarantee in this case. In the yielding version, however, the benchmark provides no such guarantee, which means the scheduler has full responsibility and any unfairness is measurable.

While this is an artificial scenario, in real life it requires only a few simple pieces. The yielding version simply creates a scenario where a thread runs uninterrupted in a saturated system and the starvation has an easily measured impact. Hence, *any* thread that runs uninterrupted for a significant time in a saturated system could lead to this kind of starvation.

7.7.1 Results

Table 7.1 shows the result for the transfer benchmark with 2 processors and all processors on the computer, where each experiment runs 100 threads per processor. Note that the results here are only meaningful as a coarse measurement of fairness, beyond which small cost differences in the runtime and concurrent primitives begin to matter. As such, data points within the same order of magnitude are considered equal. That is, the takeaway of this experiment is the presence of very large differences. The semaphore variation is denoted “Park”, where the number of threads dwindles as the new leader is acknowledged. The yielding variation is denoted “Yield”. The experiment is only run for a few and many processors since scaling is not the focus of this experiment.

The first two columns show the results for the semaphore variation on Intel. While there are some differences in latencies, with C∀ consistently the fastest and Tokio the slowest, all runtimes achieve fairly close results. Again, this experiment is meant to highlight major differences, so latencies within 10× of each other are considered equal.

Looking at the next two columns, the results for the yield variation on Intel, the story is very different. C∀ achieves better latencies, presumably due to the lack of synchronization with

Machine	Intel				AMD			
Variation	Park		Yield		Park		Yield	
processors	2	192	2	192	2	256	2	256
C∀	106 μs	19.9 ms	68.4 μs	1.2 ms	174 μs	28.4 ms	78.8 μs	1.21 ms
libfibre	127 μs	33.5 ms	DNC	DNC	156 μs	36.7 ms	DNC	DNC
Go	106 μs	64.0 ms	24.6 ms	74.3 ms	271 μs	121.6 ms	1.21 ms	117.4 ms
Tokio	289 μs	180.6 ms	DNC	DNC	157 μs	111.0 ms	DNC	DNC

Table 7.1: Transfer Benchmark on Intel and AMD

Average measurement of how long it takes for all threads to acknowledge the leader thread. For each runtime, the average is calculated over 100'000 transfers, except for Go which only has 1000 transfer (due to the difference in transfer time). DNC stands for "did not complete", meaning that after 5 seconds of a new leader being decided, some threads still had not acknowledged the new leader.

the yield. Go does complete the experiment, but with drastically higher latency: latency at 2 processors is 350× higher than C∀ and 70× higher at 192 processors. This difference is because Go has a classic work-stealing scheduler, but it adds coarse-grain preemption, which interrupts the spinning leader after a period. Neither Libfibre nor Tokio complete the experiment. Both runtimes also use classical work-stealing scheduling without preemption, and therefore, none of the work queues are ever emptied so no load balancing occurs.

Looking now at the results for the AMD architecture, the results show effectively the same story. The first two columns show all runtime obtaining results well within 10× of each other. The next two columns again show C∀ producing low latencies, while Go still has notably higher latency but the difference is less drastic on 2 processors, where it produces a 15× difference as opposed to a 100× difference on 256 processors. Neither Libfibre nor Tokio complete the experiment.

This experiment clearly demonstrates that C∀ achieves a stronger fairness guarantee. The semaphore variation serves as a control, where all runtimes are expected to transfer leadership fairly quickly. Since threads block after acknowledging the leader, this experiment effectively measures how quickly processors can steal threads from the processor running the leader. Table 7.1 shows that while Go and Tokio are slower using the semaphore, all runtimes achieve decent latency.

However, the yielding variation shows an entirely different picture. Since libfibre and Tokio have a traditional work-stealing scheduler, processors that have threads on their local queues never steal from other processors. The result is that the experiment simply does not complete for these runtimes. Without processors stealing from the processor running the leader, the experiment cannot terminate. Go manages to complete the experiment because it adds preemption on top of classic work-stealing. However, since preemption is fairly infrequent, it achieves significantly worse performance. In contrast, C∀ achieves equivalent performance in both variations, demonstrating very good fairness. Interestingly C∀ achieves better delays in the yielding version than the semaphore version, however, that is likely due to fairness being equivalent but removing

the cost of the semaphores and idle sleep.

Chapter 8

Macro-Benchmarks

The previous chapter demonstrated that the C∀ scheduler achieves its equivalent performance goal in small and controlled thread-scheduling scenarios. The next step is to demonstrate performance stays true in more realistic and complete scenarios. Therefore, this chapter exercises both thread and I/O scheduling using two flavours of web servers that demonstrate that C∀ performs competitively compared to web servers used in production environments.

Web servers are chosen because they offer fairly simple applications that perform complex I/O, both network and disk, and are useful as standalone products. Furthermore, web servers are generally amenable to parallelization since their workloads are mostly homogeneous. Therefore, web servers offer a stringent performance benchmark for C∀. Indeed, existing web servers have close to optimal performance, while the homogeneity of the workload means fairness may not be a problem. As such, these experiments should highlight the overhead due to any C∀ fairness cost in realistic scenarios.

The most obvious performance metric for web servers is throughput. This metric generally measures the speed at which the server answers and relatedly how fast clients can send requests before the server can no longer keep-up. Another popular performance metric is *tail* latency, which indicates some notion of fairness among requests across the experiment, *i.e.*, do some requests wait longer than other requests for service? Since many web applications rely on a combination of different queries made in parallel, the latency of the slowest response, *i.e.*, tail latency, can dictate a performance perception.

8.1 Memcached

Memcached [24] is an in-memory key-value store used in many production environments, *e.g.*, [17]. The Memcached server is so popular there exists a full-featured front-end for performance testing, called `mutilate` [55]. Experimenting on Memcached allows for a simple test of the C∀ runtime as a whole, exercising the scheduler, the idle-sleep mechanism, as well as the I/O subsystem for sockets. Note that this experiment does not exercise the I/O subsystem with regard to disk operations because Memcached is an in-memory server.

8.1.1 Benchmark Environment

The Memcached experiments are run on a cluster of homogeneous Supermicro SYS-6017R-TDF compute nodes with the following characteristics.

- The server runs Ubuntu 20.04.3 LTS on top of Linux Kernel 5.11.0-34.
- Each node has 2 Intel(R) Xeon(R) CPU E5-2620 v2 running at 2.10GHz.
- Each CPU has 6 cores and 2 hardware threads per core, for a total of 24 hardware threads.
- The machine is configured to run each servers on 12 dedicated hardware threads and uses 6 of the remaining hardware threads for the software interrupt handling [106], resulting in maximum CPU utilization of 75% (18 / 24 hardware threads)
- A CPU has 384 KB, 3 MB and 30 MB of L1, L2 and L3 caches, respectively.
- The compute nodes are connected to the network through a Mellanox 10 Gigabit Ethernet port.
- Network routing is performed by a Mellanox SX1012 10/40 Gigabit Ethernet switch.

8.1.2 Memcached threading

Memcached can be built to use multiple threads in addition to its `libevent` subsystem to handle requests. When enabled, the threading implementation operates as follows [66, § 16.2.2.8]:

- Threading is handled by wrapping functions within the code to provide basic protection from updating the same global structures at the same time.
- Each thread uses its own instance of the `libevent` to help improve performance.
- TCP/IP connections are handled with a single thread listening on the TCP/IP socket. Each connection is then distributed to one of the active threads on a simple round-robin basis. Each connection then operates solely within this thread while the connection remains open.
- For UDP connections, all the threads listen to a single UDP socket for incoming requests. Threads that are currently dealing with another request ignore the incoming packet. One of the remaining, non-busy, threads reads the request and sends the response. This implementation can lead to increased CPU load as threads wake from sleep to potentially process the request.

Here, Memcached is based on an event-based web server architecture [67], using kernel-level threading to run multiple largely independent event engines, and if needed, spinning up additional kernel threads to handle blocking I/O. Alternative web server architectures are:

- pipeline [94], where the event engine is subdivided into multiple stages and the stages are connected with asynchronous buffers, where the final stage has multiple threads to handle blocking I/O.
- thread-per-connection [14, 92], where each incoming connection is served by a single thread in a strict 1-to-1 pairing, using the thread stack to hold the event state and folding the event engine implicitly into the threading runtime with its nonblocking I/O mechanism.

Both pipelining and thread-per-connection add flexibility to the implementation, as the serving logic can now block without halting the event engine [39].

However, kernel-level threading in Memcached is not amenable to this work, which is based on user-level threading. While it is feasible to layer one user thread per kernel thread, it is not meaningful as it fails to exercise the user runtime; it simply adds extra scheduling overhead over the kernel threading. Hence, there is no direct way to compare Memcached using a kernel-level runtime with a user-level runtime.

Fortunately, there exists a recent port of Memcached to user-level threading based on the libfibre [49] user-level threading library. This port did all of the heavy-lifting, making it straightforward to replace the libfibre user-threading with the user-level threading in C∀. It is now possible to compare the original kernel-threading Memcached with both user-threading runtimes in libfibre and C∀.

As such, this Memcached experiment compares 3 different variations of Memcached:

- *vanilla*: the official release of Memcached, version 1.6.9.
- *fibre*: a modification of vanilla using the thread-per-connection model on top of the libfibre runtime.
- *cfa*: a modification of the fibre web server that replaces the libfibre runtime with C∀.

8.1.3 Throughput

This experiment is done by having the clients establish 15,360 total connections, which persist for the duration of the experiment. The clients then send read and write queries with 3% writes (updates), attempting to follow a desired query rate, and the server responds to the desired rate as best as possible. Figure 8.1 shows the 3 server versions at different client rates, "Target Queries Per Second", and the actual rate, "Actual QPS", for all three web servers.

Like the experimental setup in Chapter 7, each experiment is run 15 times, and for each client rate, the measured web server rate is plotted. The solid line represents the median while the dashed and dotted lines represent the maximum and minimum respectively. For rates below 500K queries per second, all three web servers match the client rate. Beyond 500K, the web servers cannot match the client rate. During this interval, vanilla Memcached achieves the highest web server throughput, with libfibre and C∀ slightly lower but very similar throughput. Overall the performance of all three web servers is very similar, especially considering that at 500K the servers have reached saturation, which is discussed more in the next section.

8.1.4 Tail Latency

Figure 8.2 shows the 99th percentile latency results for the same Memcached experiment.

Again, each experiment is run 15 times with the median, maximum and minimum plotted with different lines. As expected, the latency starts low and increases as the server gets close to saturation, at which point the latency increases dramatically because the web servers cannot keep up with the connection rate, so client requests are disproportionally delayed. Because of this dramatic increase, the Y-axis is presented using a log scale. Note that the graph shows the *target* query rate, the actual response rate is given in Figure 8.1 as this is the same underlying experiment.

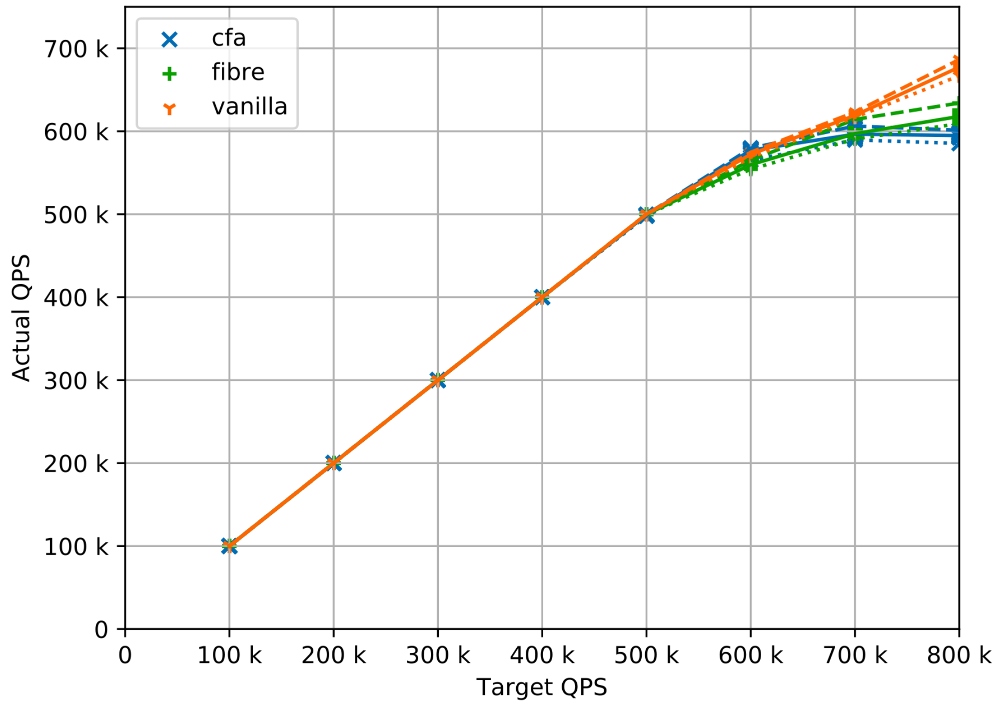


Figure 8.1: Memcached Benchmark: Throughput
Desired vs Actual query rate for 15,360 connections. Target QPS is the query rate that the clients are attempting to maintain and Actual QPS is the rate at which the server can respond.

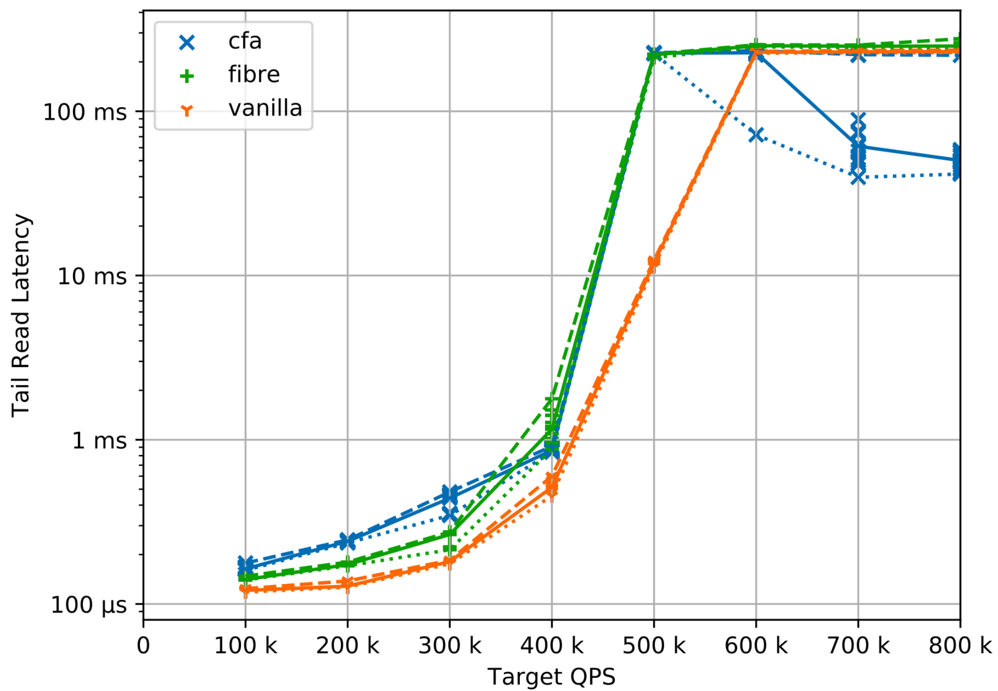


Figure 8.2: Memcached Benchmark: 99th Percentile Latency
99th Percentile of the response latency as a function of *desired* query rate for 15,360 connections.

For all three servers, the saturation point is reached before 500K queries per second, which is when throughput starts to decline among the web servers. In this experiment, all three web servers are much more distinguishable than in the throughput experiment. Vanilla Memcached achieves the lowest latency until 600K, after which all the web servers are struggling to respond to client requests. C∀ begins to decline at 600K, indicating some bottleneck after saturation. Overall, all three web servers achieve microsecond latencies and the increases in latency mostly follow each other.

8.1.5 Update rate

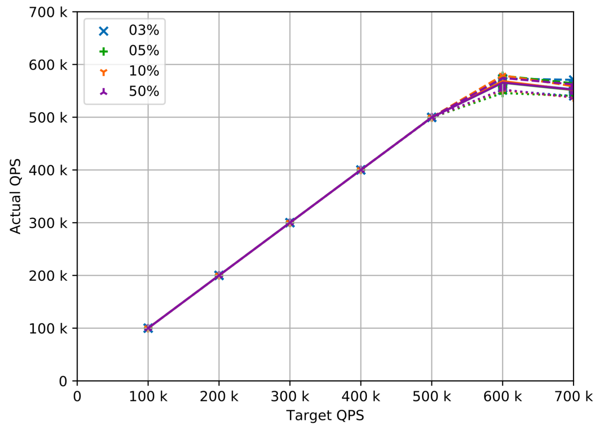
Since Memcached is effectively a simple database, the cache information can be written to concurrently by multiple queries. And since writes can significantly affect performance, it is interesting to see how varying the update rate affects performance. Figure 8.3 shows the results for the same experiment as the throughput and latency experiment but increasing the update percentage to 5%, 10% and 50%, respectively, versus the original 3% update percentage.

In the end, this experiment mostly demonstrates that the performance of Memcached is affected very little by the update rate. Indeed, since values read/written can be bigger than what can be read/written atomically, a lock must be acquired while the value is read. Hence, I believe the underlying locking pattern for reads and writes is fairly similar, if not the same. These results suggest Memcached does not attempt to optimize reads/writes using a readers-writer lock to protect each value and instead just relies on having a sufficient number of keys to limit contention. In the end, the update experiment shows that C∀ is achieving equivalent performance.

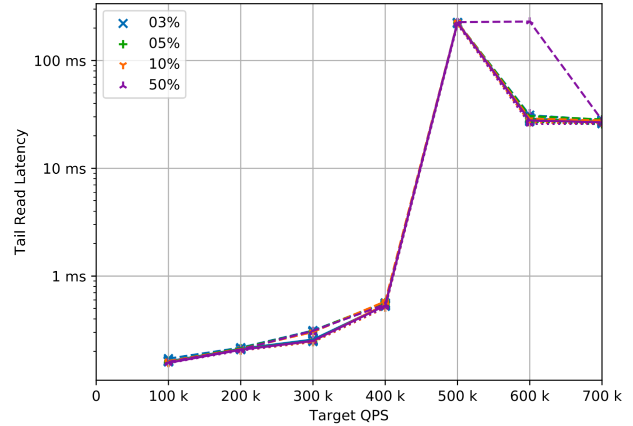
8.2 Static Web-Server

The Memcached experiment does not exercise two key aspects of the I/O subsystem: accepting new connections and interacting with disks. On the other hand, a web server servicing static web pages does stress both accepting connections and disk I/O by accepting tens of thousands of client requests per second where these requests return static data serviced from the file-system cache or disk.[1] The static web server experiment compares NGINX [64] with a custom C∀-based web server developed for this experiment. NGINX is a high-performance, *full-service*, event-driven web server. It can handle both static and dynamic web content, as well as serve as a reverse proxy and a load balancer [72]. This wealth of capabilities comes with a variety of potential configurations, dictating available features and performance. The NGINX server runs a master process that performs operations such as reading configuration files, binding to ports, and controlling worker processes. In comparison, the custom C∀ web server was developed specifically with this experiment in mind. However, nothing seems to indicate that NGINX suffers from the increased flexibility. When tuned for performance, NGINX appears to achieve the performance that the underlying hardware can achieve.

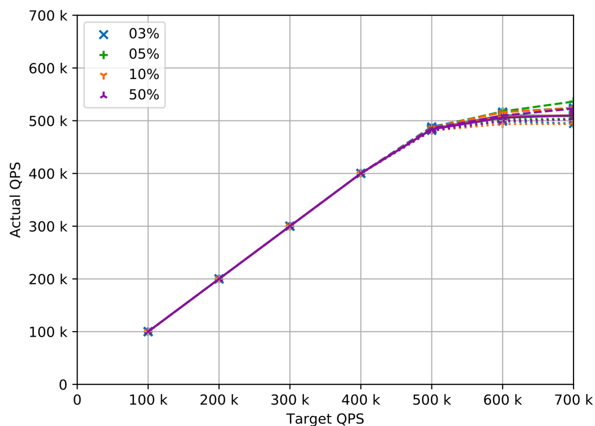
[1] web servers servicing dynamic requests, which read from multiple locations and construct a response, are not as interesting since creating the response takes more time and does not exercise the runtime in a meaningfully different way.



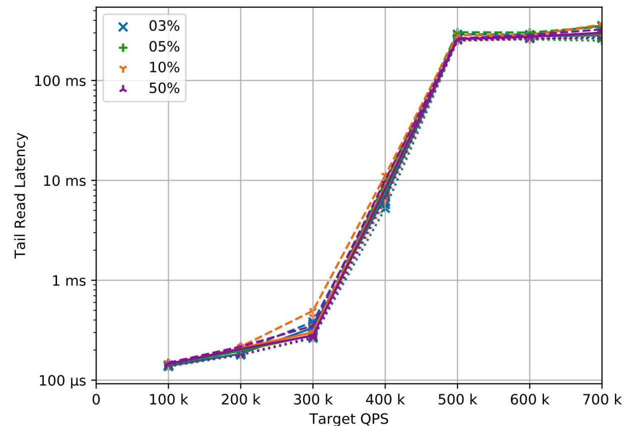
(a) C∀: Throughput



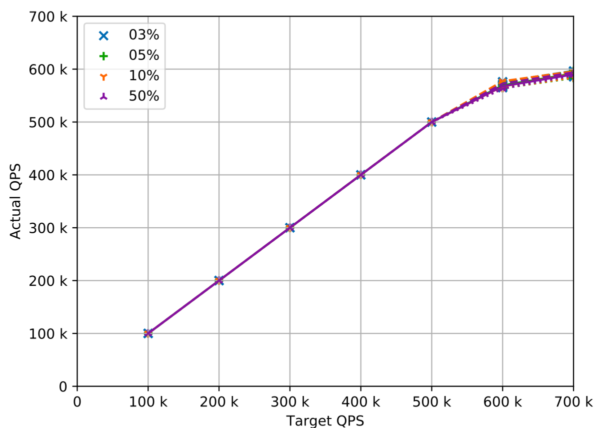
(b) C∀: Latency



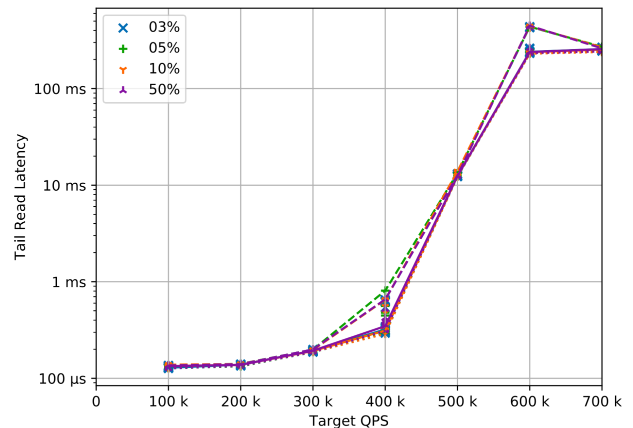
(c) LibFibre: Throughput



(d) LibFibre: Latency



(e) Vanilla: Throughput



(f) Vanilla: Latency

Figure 8.3: Throughput and Latency results at different update rates (percentage of writes).
On the left, throughput as Desired vs Actual query rate. Target QPS is the query rate that the clients are attempting to maintain and Actual QPS is the rate at which the server can respond. On the right, tail latency, *i.e.*, 99th Percentile of the response latency as a function of *desired* query rate. For throughput, higher is better, for tail-latency, lower is better. Each series represent 15 independent runs, the dashed lines are the maximums of each series while the solid lines are the median and the dotted lines are the minimums.
All runs have 15,360 client connections.

8.2.1 NGINX threading

When running as a static web server, NGINX uses an event-driven architecture to service incoming requests. Incoming connections are assigned a *stackless* HTTP state machine and worker processes can handle thousands of these state machines. For the following experiment, NGINX is configured to use `epoll` to listen for events on these state machines and have each worker process independently accept new connections. Because of the realities of Linux, (Subsection 5.1.1), NGINX also maintains a pool of auxiliary threads to handle blocking I/O. The configuration can set the number of worker processes desired, as well as the size of the auxiliary pool. However, for the following experiments, NGINX is configured to let the master process decide the appropriate number of threads.

8.2.2 C∀ web server

The C∀ web server is a straightforward thread-per-connection web server, where a fixed number of threads are created upfront. Each thread calls `accept`, through `io_uring`, on the listening port and handles the incoming connection once accepted. Most of the implementation is fairly straightforward; however, the inclusion of file I/O found an `io_uring` problem that required an unfortunate workaround.

Normally, web servers use `sendfile` [78] to send files over a socket because it performs a direct move in the kernel from the file-system cache to the NIC, eliminating reading/writing the file into the web server. While `io_uring` does not support `sendfile`, it does support `splice` [79], which is strictly more powerful. However, because of how Linux implements file I/O, see Subsection 5.1.1, `io_uring` must delegate splice calls to worker threads *inside* the kernel. As of Linux 5.13, `io_uring` had no mechanism to restrict the number of worker threads, and therefore, when tens of thousands of splice requests are made, it correspondingly creates tens of thousands of internal kernel-level threads. Such a high number of kernel-level threads slows Linux significantly. Rather than abandon the experiment, the C∀ web server was switched to `sendfile`.

Starting with *blocking* `sendfile`, C∀ achieves acceptable performance until saturation is reached. At saturation, latency increases and client connections begin to timeout. As these clients close their connection, the server must close its corresponding side without delay so the OS can reclaim the resources used by these connections. Indeed, until the server connection is closed, the connection lingers in the CLOSE-WAIT TCP state [70] and the TCP buffers are preserved. However, this poses a problem using blocking `sendfile` calls: when `sendfile` blocks, the processor rather than the thread blocks, preventing other connections from closing their sockets. The call can block if there is insufficient memory, which can be caused by having too many connections in the CLOSE-WAIT state.[2] This effect results in a negative feedback loop where more timeouts lead to more `sendfile` calls running out of resources.

Normally, this problem is addressed by using `select`/`epoll` to wait for sockets to have sufficient resources. However, since `io_uring` does not support `sendfile` but does respect non-

[2] `sendfile` can always block even in nonblocking mode if the file to be sent is not in the file-system cache, because Linux does not provide nonblocking disk I/O.

blocking semantics, marking all sockets as non-blocking effectively circumvents the `io_uring` subsystem entirely: all calls simply immediately return `EAGAIN` and all asynchronicity is lost.

Switching the entire C∀ runtime to `epoll` for this experiment is unrealistic and does not help in the evaluation of the C∀ runtime. For this reason, the C∀ web server sets and resets the `O_NONBLOCK` flag before and after any calls to `sendfile`. However, when the nonblocking `sendfile` returns `EAGAIN`, the C∀ server cannot block the thread because its I/O subsystem uses `io_uring`. Therefore, the thread spins performing the `sendfile`, yields if the call returns `EAGAIN` and retries in these cases.

Interestingly, Linux 5.15 `io_uring` introduces the ability to limit the number of worker threads that are created through the `IORING_REGISTER_IOWQ_MAX_WORKERS` option. Presumably, this limit would prevent the explosion of kernel-level threads, which justified using `sendfile` over `io_uring` and `splice`. However, recall from Section 5.2.1 that `io_uring` maintains two pools of workers: bounded workers and unbounded workers. For a web server, the unbounded workers should handle accepts and reads on sockets, and the bounded workers should handle reading files from disk. This setup allows fine-grained control over the number of workers needed for each operation type and presumably leads to good performance.

However, `io_uring` must contend with another reality of Linux: the versatility of `splice`. Indeed, `splice` can be used both for reading and writing to or from any type of file descriptor. This generality makes it ambiguous which pool `io_uring` should delegate `splice` calls to. In the case of splicing from a socket to a pipe, `splice` behaves like an unbounded operation, but when splicing from a regular file to a pipe, `splice` becomes a bounded operation. To make things more complicated, `splice` can read from a pipe and write to a regular file. In this case, the read is an unbounded operation but the write is a bounded one. This leaves `io_uring` in a difficult situation where it can be very difficult to delegate splice operations to the appropriate type of worker. Since there is little or no context available to `io_uring`, it seems to always delegate `splice` operations to the unbounded workers. This decision is unfortunate for this specific experiment since it prevents the web server from limiting the number of parallel calls to `splice` without affecting the performance of `read` or `accept`. For this reason, the `sendfile` approach described above is still the most performant solution in Linux 5.15.

One possible workaround is to create more `io_uring` instances so `splice` operations can be issued to a different instance than the `read` and `accept` operations. However, I do not believe this solution is appropriate in general; it simply replaces my current web server hack with a different, equivalent hack.

8.2.3 Benchmark Environment

Unlike the Memcached experiment, the web server experiment is run on a heterogeneous environment.

- The server runs Ubuntu 20.04.4 LTS on top of Linux Kernel 5.13.0-52.
- The server computer has four AMD Opteron™Processor 6380 with 16 cores running at 2.5GHz, for a total of 64 hardware threads.
- The computer is booted with only 8 CPUs enabled, which is sufficient to achieve line rate.

- Both servers are setup with enough parallelism to achieve 100% CPU utilization, which happens at higher request rates.
- Each CPU has 64 KB, 256 KiB and 8 MB of L1, L2 and L3 caches respectively.
- The computer is booted with only 25GB of memory to restrict the file-system cache.

There are 8 client machines.

- A client runs a 2.6.11-1 SMP Linux kernel, which permits each client load generator to run on a separate CPU.
- It has two 2.8 GHz Xeon CPUs, and four one-gigabit Ethernet cards.
- Network routing is performed by an HP 2530 10 Gigabit Ethernet switch.
- A client machine runs two copies of the workload generator.

The clients and network are sufficiently provisioned to drive the server to saturation and beyond. Hence, any server effects are attributable solely to the runtime system and web server. Finally, without restricting the server hardware resources, it is impossible to determine if a runtime system or the web server using it has any specific design restrictions, *e.g.*, using space to reduce time. Trying to determine these restrictions with large numbers of processors or memory simply means running equally large experiments, which take longer and are harder to set up.

8.2.4 Throughput

To measure web server throughput, the server computer is loaded with 21,600 files, sharded across 650 directories, occupying about 2.2GB of disk, distributed over the server's RAID-5 4-drives to achieve high throughput for disk I/O. The clients run httperf [60] to request a set of static files. The httperf load generator is used with session files to simulate a large number of users and to implement a partially open-loop system. This permits httperf to produce overload conditions, generate multiple requests from persistent HTTP/1.1 connections, and include both active and inactive off periods to model browser processing times and user think times [19].

The experiments are run with 16 clients, each running a copy of httperf (one copy per CPU), requiring a set of 16 log files with requests conforming to a Zipf distribution. This distribution is representative of users accessing static data through a web browser. Each request reads a file name from its trace, establishes a connection, performs an HTTP GET request for the file name, receives the file data, closes the connection, and repeats the process. Some trace elements have multiple file names that are read across a persistent connection. A client times out if the server does not complete a request within 10 seconds.

An experiment consists of running a server with request rates ranging from 10,000 to 70,000 requests per second; each rate takes about 5 minutes to complete. There are 20 seconds of idle time between rates and between experiments to allow connections in the TIME-WAIT state to clear. Server throughput is measured both at peak and after saturation (*i.e.*, after peak). Peak indicates the level of client requests the server can handle and after peak indicates if a server degrades gracefully. Throughput is measured by aggregating the results from httperf for all the clients.

Table 8.1: Cumulative memory for requests by file size

% Requests	10	30	50	70	80	90	**95**	100
Memory (MB)	0.5	1.5	8.4	12.2	20.1	94.3	**126.5**	2,291.6
File Size (B)	409	716	4,096	5,120	7,168	40,960	**51,200**	921,600

This experiment can be done for two workload scenarios by reconfiguring the server with different amounts of memory: 25 GB and 2.5 GB. The two workloads correspond to in-memory and disk-I/O respectively. Due to the Zipf distribution, only a small amount of memory is needed to service a significant percentage of requests. Table 8.1 shows the cumulative memory required to satisfy the specified percentage of requests; e.g., 95% of the requests come from 126.5 MB of the file set and 95% of the requests are for files less than or equal to 51,200 bytes. Interestingly, with 2.5 GB of memory, significant disk-I/O occurs.

Figure 8.4 shows the results comparing C∀ to NGINX in terms of throughput. These results are fairly straightforward. Both servers achieve the same throughput until around 57,500 requests per second. Since the clients are asking for the same files, the fact that the throughput matches exactly is expected as long as both servers are able to serve the request rate. Once the saturation point is reached, both servers are still very close. NGINX achieves slightly better throughput. However, Figure 8.4b shows the rate of errors, a gross approximation of tail latency, where C∀ achieves notably fewer errors once the servers reach saturation. This suggests C∀ is slightly fairer with less throughput, while NGINX sacrifices fairness for more throughput. This experiment demonstrates that the C∀ web server is able to match the performance of NGINX up to and beyond the saturation point of the machine.

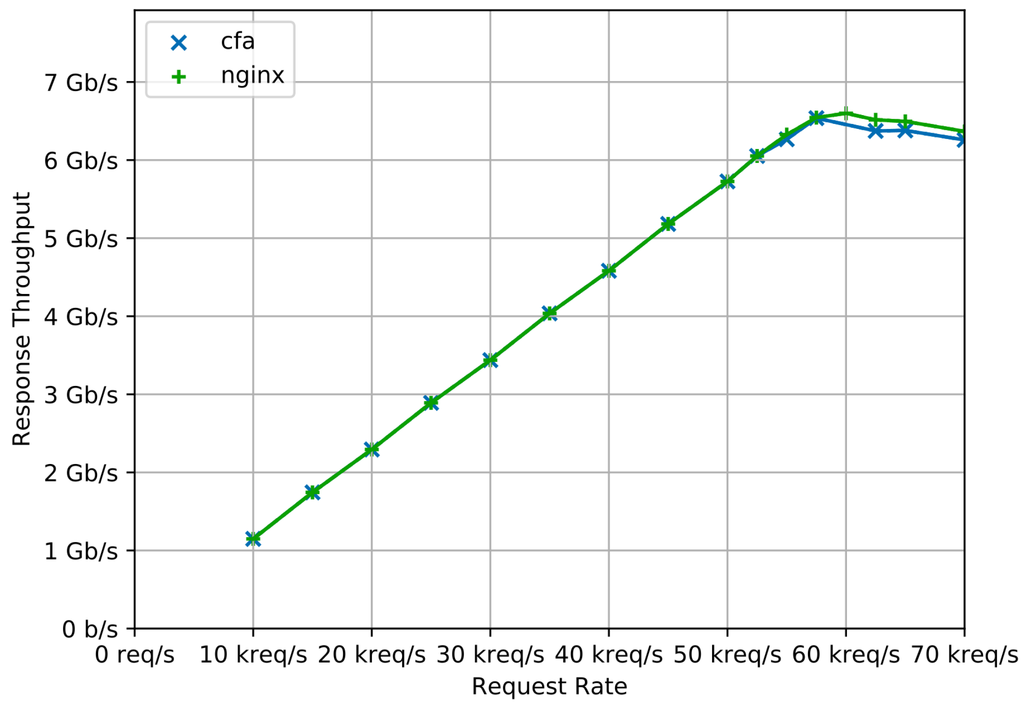
8.2.5 Disk Operations

With 25GB of memory, the entire experimental file-set plus the web server and OS fit in memory. If memory is constrained, the OS must evict files from the file cache, which causes `sendfile` to read from disk.[3] web servers can behave very differently once file I/O begins and increases. Hence, prior work [38] suggests running both kinds of experiments to test overall web server performance.

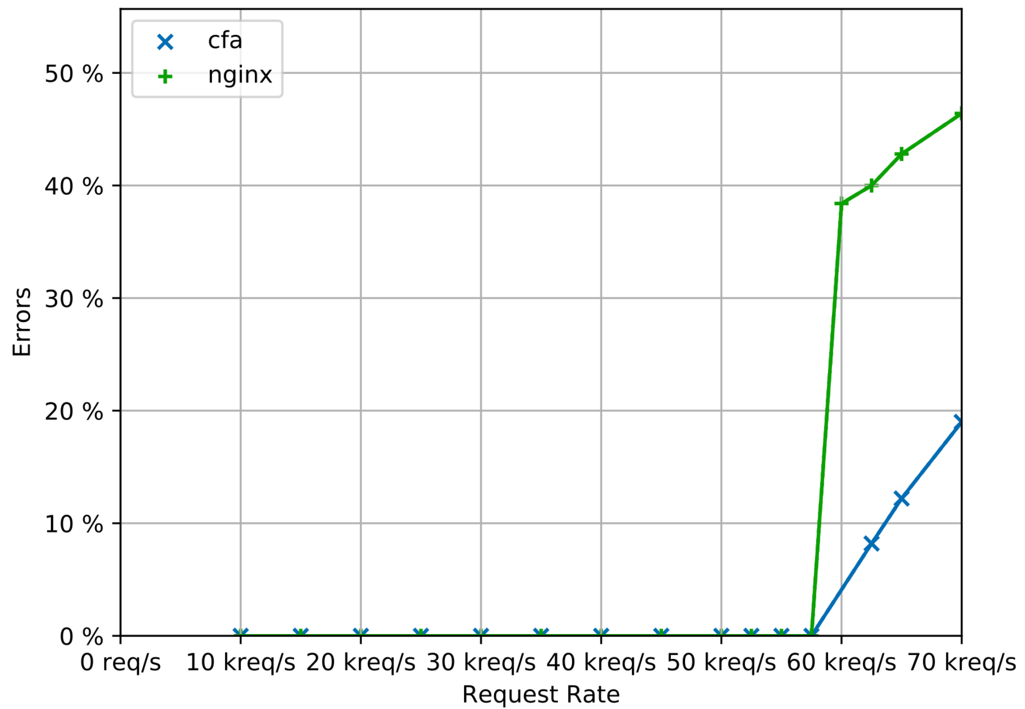
However, after reducing memory to 2.5GB, the problem with `splice` and `io_uring` rears its ugly head again. Indeed, in the in-memory configuration, replacing `splice` with calls to `sendfile` works because the bounded side basically never blocks. Like `splice`, `sendfile` is in a situation where the read side requires bounded blocking, *e.g.*, reading from a regular file, while the write side requires unbounded blocking, *e.g.*, blocking until the socket is available for writing. The unbounded side can be handled by yielding when it returns `EAGAIN`, as mentioned above, but this trick does not work for the bounded side. The only solution for the bounded side is to spawn more threads and let these handle the blocking.

Supporting this case in the web server would require creating more processors or creating a dedicated thread pool. However, I felt this kind of modification moves too far away from my goal

[3]For the in-memory experiments, the file-system cache was warmed by running an experiment three times before measuring started to ensure all files are in the file-system cache.



(a) Throughput



(b) Rate of Errors

Figure 8.4: Static web server Benchmark: Throughput
Throughput vs request rate for short-lived connections.

of evaluating the C∀ runtime, *i.e.*, it begins writing another runtime system; hence, I decided to forgo experiments on low-memory performance.

Part IV

Conclusion & Annexes

Chapter 9

Conclusion

Building the C∀ runtime has been a challenging project. The work was divided between high-level concurrency design and a user-level threading runtime (Masters' thesis), and low-level support of the user-level runtime using OS kernel threading and its (multiple) I/O subsystems (Ph.D. thesis). Because I am the main developer for both components of this project, there is strong continuity across the design and implementation. This continuity provides a consistent approach to advanced control flow and concurrency, with easier development, management and maintenance of the runtime in the future.

I believed my Masters' work would provide the background to make the Ph.D. work reasonably straightforward. However, I discovered two significant challenges.

First, modern symmetric multiprocessing CPUs have significant performance penalties for communication, often cache-related. An SQMS scheduler (see Section 1.1), with its processor-shared ready-queue, has perfect load-balancing but poor affinity resulting in high communication across processors. An MQMS scheduler, with its processor-specific ready-queues, has poor load-balancing but perfect affinity often resulting in significantly reduced communication. However, implementing fairness for an MQMS scheduler is difficult, since fairness requires processors to be aware of each other's ready-queue progress, *i.e.*, communicated knowledge. For balanced workloads with little or no data sharing, *i.e.*, embarrassingly parallel, an MQMS scheduler is near optimal, *e.g.*, a state-of-the-art work-stealing scheduler. For these kinds of fair workloads, adding fairness must be low-cost to hide the communication costs needed for global ready-queue progress or performance suffers. While I was aware of these realities, I underestimated how little performance margin there is for communication. Several of my attempts at building a fair scheduler compared poorly to work-stealing schedulers because of the thin communication margin.

Second, the kernel locking, threading, and I/O in the Linux operating system offer very little flexibility and are not designed to facilitate user-level threading. There are multiple concurrency aspects in Linux that require carefully following a strict procedure to achieve acceptable performance. To be fair, many of these concurrency aspects were designed 30-40 years ago, when there were few multiprocessor computers and concurrency knowledge was just developing. Unfortunately, little has changed in the intervening years.

Also, my decision to use `io_uring` was both positive and negative. The positive is that `io_uring` supports the panoply of I/O mechanisms in Linux; hence, the C∀ runtime uses one

I/O mechanism to provide non-blocking I/O, rather than using `select` to handle TTY I/O, `epoll` to handle network I/O, and managing a thread pool to handle disk I/O. Merging all these different I/O mechanisms into a coherent scheduling implementation would require much more work than what is present in this thesis, as well as detailed knowledge of multiple I/O mechanisms. The negative is that `io_uring` is new and developing. As a result, there is limited documentation, few places to find usage examples, and multiple errors that required workarounds.

Given what I now know about `io_uring`, I would say it is insufficiently coupled with the Linux kernel to properly handle non-blocking I/O. It does not seem to reach deep into the kernel's handling of I/O, and as such it must contend with the same realities that users of `epoll` must contend with. Specifically, in cases where `O_NONBLOCK` behaves as desired, operations must still be retried. Preserving the illusion of asynchronicity requires delegating these operations to kernel threads. This requirement is also true of cases where `O_NONBLOCK` does not prevent blocking. Spinning up internal kernel threads to handle blocking scenarios is what developers already do outside of the kernel, and managing these threads adds a significant burden to the system. Nonblocking I/O should not be handled in this way. Presumably, this is better handled by Windows's "overlapped I/O", however porting C∀ to Windows is far beyond the scope of this work.

9.1 Goals

This work focuses on efficient and fair scheduling of the multiple CPUs, which are ubiquitous on all modern computers. The levels of indirection to the CPUs are:

- The C∀ presentation of concurrency through multiple high-level language constructs.
- The OS presentation of concurrency through multiple kernel threads within an application.
- The OS and library presentation of disk and network I/O, and many secondary library routines that directly and indirectly use these mechanisms.

The key aspect of all of these mechanisms is that control flow can block, which immediately hinders any level above from making scheduling decisions as a result. Fundamentally, scheduling needs to understand all the mechanisms used by threads that affect their state changes.

The underlying goal of this thesis is scheduling the complex hardware components that make up a computer to provide good utilization and fairness. However, direct hardware scheduling is only possible in the OS. Instead, this thesis is performing arms-length application scheduling of the hardware components through a set of OS interfaces that indirectly manipulate the hardware components. This can quickly lead to tensions when the OS interface has different use cases in mind.

As C∀ aims to increase productivity and safety of C, while maintaining its performance, this places a huge burden on the C∀ runtime to achieve these goals. Productivity and safety manifest in removing scheduling pitfalls from the efficient usage of the threading runtime. Performance manifests in making efficient use of the underlying kernel threads that provide indirect access to the CPUs.

This thesis achieves its stated contributions by presenting:

1. A scalable low-latency scheduler that offers improved starvation prevention (progress guarantee) compared to other state-of-the-art schedulers, including NUMA awareness.
2. The scheduler demonstrates a core algorithm that provides increased fairness through helping, as well as optimizations which virtually remove the cost of this fairness.
3. An implementation of user-level I/O blocking is incorporated into the scheduler, which achieves the same performance and fairness balance as the scheduler itself.
4. These core algorithms are further extended with a low-latency idle-sleep mechanism, which allows the C∀ runtime to stay viable for workloads that do not consistently saturate the system.

Finally, the complete scheduler is fairly simple with low-cost execution, meaning the total cost of scheduling during thread state changes is low.

9.2 Future Work

While the C∀ runtime achieves a better compromise than other schedulers, in terms of performance and fairness, I believe further improvements can be made to reduce or eliminate the few cases where performance does deteriorate. Fundamentally, achieving performance and starvation freedom will always be goals with opposing needs even outside of scheduling algorithms.

9.2.1 Idle Sleep

A difficult challenge, not fully addressed in this thesis, is idle sleep. While a correct and somewhat low-cost idle-sleep mechanism is presented, several of the benchmarks show notable performance degradation when too few threads are present in the system. The idle sleep mechanism could therefore benefit from a reduction of spurious cases of sleeping. Furthermore, this thesis did not present any heuristic for when processors should be put to sleep and when processors should be woken up. While relaxed timestamps and topology awareness made notable performance improvements, neither of these techniques are used for the idle-sleep mechanism.

Here are opportunities where these techniques could be used:

- The mechanism uses a handshake between notification and sleep to ensure that no thread is missed.
- The handshake correctness is critical when the last processor goes to sleep but could be relaxed when several processors are awake.
- Furthermore, organizing the sleeping processors as a LIFO stack makes sense to keep cold processors as cold as possible, but it might be more appropriate to attempt to keep cold CPU sockets instead.

However, using these techniques would require significant investigation. For example, keeping a CPU socket cold might be appropriate for power consumption reasons but can affect overall memory bandwidth. The balance between these approaches is not obvious. I am aware there is a host of low-power research that could be tapped here.

9.2.2 CPU Workloads

A performance consideration related to idle sleep is cpu utilization, *i.e.*, how easy is it CPU utilization generally becomes an issue for workloads that are compute bound but where the dependencies among threads can prevent the scheduler from easily. Examining such workloads in the context of scheduling would be interesting. However, such workloads are inherently more complex than applications examined in this thesis, and as such warrant it's own work.

9.2.3 Hardware

One challenge that needed to be overcome for this thesis is that the modern x86-64 processors have very few tools to implement fairness. Processors attempting to help each other inherently cause cache-coherence traffic. However, as mentioned in Section 4.4, relaxed requirements mean this traffic is not necessarily productive. In cases like this one, there is an opportunity to improve performance by extending the hardware.

Many different extensions are suitable here. For example, when attempting to read remote timestamps for helping, it would be useful to allow cancelling the remote read if it leads to significant latency. If the latency is due to a recent cache invalidation, it is unlikely the timestamp is old and that helping is needed. As such, simply moving on without the result is likely to be acceptable. Another option is to read multiple memory addresses and only wait for *one of* these reads to retire. This approach has a similar effect, where cache lines with more traffic are waited on less often. In both of these examples, some care is needed to ensure that reads to an address *sometimes* retire.

Note that this idea is similar to *Hardware Transactional Memory* [110], which allows groups of instructions to be aborted and rolled back if they encounter memory conflicts when being retired. However, I believe this feature is generally aimed at large groups of instructions. A more fine-grained approach may be more amenable by carefully picking which aspects of an algorithm require exact correctness and which do not.

References

[1] https://www.baeldung.com/openjdk-project-loom.

[2] https://kotlinlang.org/docs/multiplatform-mobile-concurrency-and-coroutines.html.

[3] https://www.baeldung.com/java-fork-join.

[4] CFS scheduler - the linux kernel documentation. https://www.kernel.org/doc/html/latest/-scheduler/sched-design-CFS.html.

[5] Reworking CFS load balancing. https://lwn.net/Articles/793427, 2019.

[6] *SCHED(7) - Linux Programmer's Manual*, august 2019.

[7] *KQUEUE(2) - FreeBSD System Calls Manual*, may 2020.

[8] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002.

[9] *accept(2) Linux User's Manual*, March 2019.

[10] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In Guy E. Blelloch and Peter Sanders, editors, *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 84–95. ACM, 2014.

[11] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Helper locks for fork-join parallel programming. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 245–256. ACM, 2010.

[12] Susanne Albers and Antonios Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. In *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1266–1285, January 2012.

[13] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. Relaxed schedulers can efficiently parallelize iterative algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 377–386, 2018.

[14] The Apache web server. http://httpd.apache.org. [Online; accessed 6-June-2022].

[15] Apple Inc. *Mach Scheduling and Thread Interfaces - Kernel Programming Guide*. https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/-KernelProgramming/scheduler/scheduler.html.

[16] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[18] Jens Axboe. Efficient io with io_uring. https://kernel.dk/io_uring.pdf, March 2019.

[19] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of ACM SIGMETRICS 1998*, Madison, Wis., 1998.

[20] Michael A. Bender and Michael O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory Comput. Syst.*, 35(3):289–304, 2002.

[21] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5):1260–1279, 2003.

[22] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[23] Robert D. Blumofe. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 356–368. IEEE Computer Society, 1994.

[24] Memcached. http://httpd.apache.org, 2003. [Online; accessed 6-June-2022].

[25] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In Arvind and Jack B. Dennis, editors, *Proceedings of the 1981 conference on Functional programming languages and computer architecture, FPCA 1981, Wentworth, New Hampshire, USA, October 1981*, pages 187–194. ACM, 1981.

[26] C∀ Features. https://plg.uwaterloo.ca/∼cforall/features.

[27] Kate Chase and Mark E. Russinovich. *Windows Internals*, chapter Processes, Threads, and Jobs in the Windows Operating System. Developer Reference. Microsoft Press, 5th edition edition, June 2009.

[28] Richard Cole and Vijaya Ramachandran. Analysis of randomized work stealing with false sharing. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 985–998. IEEE Computer Society, 2013.

[29] Jonathan Corbet. Per-entity load tracking. *LWN article, available at: https://lwn.net/-Articles/531853*, 2013.

[30] Thierry Delisle. Scheduling benchmarks. https://github.com/cforall/Scheduling-Benchmarks_PhD22.

[31] Thierry Delisle. Concurrency in C∀. Master's thesis, School of Computer Science, University of Waterloo, 2018. https://uwspace.uwaterloo.ca/handle/10012/12888.

[32] Thierry Delisle and Peter A. Buhr. Advanced control-flow and concurrency in C∀. *Softw. Pract. Exper.*, 51(5):1005–1042, May 2021. https://onlinelibrary.wiley.com/doi/10.1002/-spe.2925.

[33] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Perform. Evaluation*, 6(1):53–68, 1986.

[34] *epoll(7) Linux User's Manual*, March 2019.

[35] *eventfd(2) Linux User's Manual*, March 2019.

[36] Nicolas Gast and Bruno Gaujal. A mean field model of work stealing in large-scale systems. In Vishal Misra, Paul Barford, and Mark S. Squillante, editors, *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010*, pages 13–24. ACM, 2010.

[37] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic programming. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[38] Ashif Harji. *Performance Comparison of Uniprocessor and Multiprocessor Web Server Architectures*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, February 2010. http://uwspace.uwaterloo.ca/bitstream/10012/5040/1/Harji_thesis.pdf.

[39] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. Comparing high-performance multi-core web-server architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 1:1–1:12, New York, NY, USA, June 2012. ACM.

[40] IBM. Serially reusable programs. https://www.ibm.com/docs/en/ztpf/1.1.0.15?-topic=structures-serially-reusable-programs, March 2021.

[41] IEEE and The Open Group. Pthread.h, specifications issue 7, IEEE std 1003.1-2017. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html, 2018.

[42] IEEE and The Open Group. *1003.1 Standard for Information Technology – Portable Operating System Interface (POSIX), Base Specifications, Issue 7*, 2017. https://-pubs.opengroup.org/onlinepubs/9699919799.

[43] Apple Inc. Grand central dispatch, a better way to do multicore. Technical report, August 2009. [Online; accessed 5-August-2022].

[44] Apple Inc. *Grand Central Dispatch*, 2022. [Online; accessed 5-August-2022].

[45] Intel®. *Scheduling Algorithm - Intel® Threading Building Blocks Developer Reference*.

[46] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3B: System Programming Guide, Part 2. Intel®, 2016.

[47] Marty Kalin. CFS: Completely fair process scheduling in linux. https://opensource.com/article/19/2/fair-scheduling-linux, February 2019.

[48] Martin Karsten. libfibre: User-Level Threading Runtime. https://git.uwaterloo.ca/mkarsten/libfibre. [Online; accessed 2020-04-15].

[49] Martin Karsten and Saman Barghi. User-level threading: Have your cake and eat it too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1):17:1–17:30, 2020.

[50] Martin Karsten and Saman Barghi. User-level Threading: Have Your Cake and Eat It Too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), June 2020.

[51] Ericsson AB Kenneth Lundin. Inside the erlang vm. In *Erlang User Conference*, 2008.

[52] The Go Programming Language. cgo. . [Online; accessed 5-August-2022].

[53] The Go Programming Language. Github - the go programming language. https://github.com/golang/go.

[54] The Go Programming Language. src/runtime/preempt.go. . [Online; accessed 5-August-2022].

[55] Jacob Leverich. Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate.

[56] libuv team. libuv: Asynchronous i/o made simple. https://libuv.org/. [Online; accessed 5-August-2022].

[57] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 1:1–1:16. ACM, 2016.

[58] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, volume 625. Citeseer, 2011.

[59] Ravi Mirchandaney, Donald F. Towsley, and John A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *J. Parallel Distributed Comput.*, 9(4):331–346, 1990.

[60] David Mosberger and Tai Jin. httperf tool for measuring web server performance. *ACM SIGMETRICS*, 26(3):31–37, 1998.

[61] Aaron Moss, Robert Schluntz, and Peter A. Buhr. C∀ : Adding modern programming language features to C. *Softw. Pract. Exper.*, 48(12):2111–2146, December 2018. http://dx.doi.org/10.1002/spe.2624.

[62] Randall Munroe. 908: The cloud. https://xkcd.com/908/, June 2011. [Online; accessed 25-August-2022].

[63] Randall Munroe. 2318: Dynamic entropy. https://xkcd.com/2318/, June 2020. [Online; accessed 10-June-2020].

[64] NGINX. https://www.nginx.com.

[65] *open(2) Linux User's Manual*, February 2020.

[66] Oracle. Mysql 5.6 reference manual including mysql ndb cluster 7.3-7.4 reference guide. https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/ha-memcached-using-threads.html. [Online; accessed 5-August-2022].

[67] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, California, U.S.A., June 1999. USENIX Association.

[68] Parallel Universe. *Quasar Core - Quasar User Manual*.

[69] *poll(2) Linux User's Manual*, July 2019.

[70] Jon Postel. Transmission control protocol. Technical report, 1981.

[71] Chet Ramey and Brian Fox. *Bash Reference Manual*, December 2020.

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

[73] Haris Ribic and Yu David Liu. Energy-efficient work-stealing language runtimes. *ACM SIGARCH Computer Architecture News*, 42(1):513–528, 2014.

[74] Jeff Roberson. ULE: A modern scheduler for freebsd. In Gregory Neil Shapiro, editor, *Proceedings of BSDCon 2003, San Mateo, California, USA, September 8-12, 2003*, pages 17–28. USENIX, 2003.

[75] M.E. Russinovich, D.A. Solomon, and A. Ionescu. *Windows Internals*. Developer Reference Series. Microsoft Press, 2009.

[76] Benoit Schillings. Be engineering insights: Benaphores. *Be Newsletters*, 1(26), 1996.

[77] *select(2) Linux User's Manual*, March 2019.

[78] *sendfile(2) Linux User's Manual*, September 2017.

[79] *splice(2) Linux User's Manual*, May 2019.

[80] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distributed Syst.*, 4(2):131–143, 1993.

[81] Mark S. Squillante and Randolph D. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. In Tom W. Keller, editor, *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems, San Diego, California, USA, May 21-24, 1991*, pages 143–155. ACM, 1991.

[82] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. On the efficiency of localized work stealing. *Inf. Process. Lett.*, 116(2):100–106, 2016.

[83] *Synchronous and Asynchronous IO*, March 2021. [Online; accessed 5-August-2022].

[84] *TaskSettings.Priority property*, September 2020. [Online; accessed 5-August-2022].

[85] Marc Tchiboukdjian, Nicolas Gast, and Denis Trystram. Decentralized list scheduling. *Ann. Oper. Res.*, 207(1):237–259, 2013.

[86] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*, volume 6507 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2010.

[87] Threading Model. Thread (computing). https://en.wikipedia.org/wiki/Thread_(computing).

[88] Tokio. Tokio asynchronous runtime for Rust. https://tokio.rs.

[89] Christopher Torng, Moyang Wang, and Christopher Batten. Asymmetry-aware work-stealing runtimes. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 40–52. IEEE, 2016.

[90] Linus Torvalds. Re: [patch 09/13] aio: add support for async openat(). https://lwn.net/Articles/671657, January 2016. [Online; accessed 6-June-2022].

[91] BRWACRR Vikranth, Rajeev Wankar, and C Raghavendra Rao. Topology aware task stealing for on-chip numa multi-core processors. *Procedia Computer Science*, 18:379–388, 2013.

[92] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[93] Dmitry Vyukov. Go scheduler: Implementing language with lightweight concurrency. In *Hydra*, 2019.

[94] 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.

[95] Wikipedia contributors. Explicit parallelism — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Explicit_parallelism, 2017. [Online; accessed 23-October-2020].

[96] Wikipedia contributors. Control theory — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Task_parallelism, 2020. [Online; accessed 22-October-2020].

[97] Wikipedia contributors. Futures and promises — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Futures_and_promises, 2020. [Online; accessed 9-February-2021].

[98] Wikipedia contributors. Implicit parallelism — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Implicit_parallelism, 2020. [Online; accessed 23-October-2020].

[99] Wikipedia contributors. Linear congruential generator — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Linear_congruential_generator, 2020. [Online; accessed 2-January-2021].

[100] Wikipedia contributors. Task parallelism — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Control_theory, 2020. [Online; accessed 22-October-2020].

[101] Wikipedia contributors. Java native interface — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Java_Native_Interface, 2021. [Online; accessed 5-August-2022].

[102] Wikipedia contributors. Readers-writer lock — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Readers-writer_lock, 2021. [Online; accessed 12-April-2022].

[103] Wikipedia contributors. Bin packing problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Bin_packing_problem, 2022. [Online; accessed 29-June-2022].

[104] Wikipedia contributors. Bin packing problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average, 2022. [Online; accessed 5-August-2022].

[105] Wikipedia contributors. Expected value — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Expected_value, 2022. [Online; accessed 22-November-2022].

[106] Wikipedia contributors. Interrupt — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Interrupt, 2022. [Online; accessed 24-November-2022].

[107] Wikipedia contributors. Non-blocking algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Non-blocking_algorithm, 2022. [Online; accessed 22-November-2022].

[108] Wikipedia contributors. Read-copy-update — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Linear_congruential_generator, 2022. [Online; accessed 12-April-2022].

[109] Wikipedia contributors. Time stamp counter — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Time_Stamp_Counter, 2022. [Online; accessed 14-November-2022].

[110] Wikipedia contributors. Transactional memory — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Zipf%27s_law, 2022. [Online; accessed 7-September-2022].

[111] Jixiang Yang and Qingbi He. Scheduling parallel computations by work stealing: A survey. *Int. J. Parallel Program.*, 46(2):173–197, 2018.

Glossary

Thread Blocking Thread blocking means taking a running thread off a CPU. Unless no other thread is ready, this action is immediately followed by running another thread.

Synonyms : Parking. 40, 50, 57, 58

Threads Migration Migration refers to the idea of an thread running on a different processor than the last time it was run. It is generally preferable to minimize migration as it incurs cost but any load balancing among processor requires some amount of migration.

Synonyms : None. 10, 13, 23, 28, 40

Hardware Threading Threads representing the underlying hardware, *e.g.*, a CPU core or hyper-thread, if the hardware supports multiple threads of execution per core. The number of hardware threads present is fixed on any given computer.

Synonyms : Core, Hyper-Thread, Processing Unit, CPU. 16, 22, 24, 26, 27, 29, 32, 39, 42, 44, 49, 56, 63, 65, 77, 83, 100, 101

Job Unit of work, often sent to a thread pool or worker pool to be executed. Has neither its own stack nor its own thread of execution.

Synonyms : Tasks. 101

Kernel-Level Thread Threads created and managed inside kernel space. Each kernel thread has its own stack and its own thread of execution. Kernel-level threads are owned, managed and scheduled by the underlying operating system.

Synonyms : OS threads, Hardware threads, Physical threads. iv, 3, 5, 15, 17, 18, 33–37, 39, 49, 50, 52, 60, 77, 78, 82, 83, 101

Processor Entity that executes a thread, *i.e.*, the resource being scheduled by the scheduler. In kernel-level threading, threads are kernel threads and processors are the hardware threads on which the kernel threads are scheduled. In user-level threading and thread pools, processors are kernel threads.

Synonyms : Server, Worker. 7, 10–13, 15–17, 21–27, 29, 30, 32, 33, 36, 37, 40, 41, 43, 44, 47–54, 56–71, 73, 74, 82, 85, 89, 91, 92, 100, 101

Ready Queue Data structure holding threads that are ready to run. Often a FIFO queue for fairness, but can take many different forms, *e.g.*, binary tree and priority queue are also common. 5

Remote Memory Reference A memory reference to an address not in the current hardware thread's cache is a remote reference. Memory references that *are* in the current hardware thread's cache is a *local* memory reference. For example, a cache line that must be updated from the any cache on another socket, or from RAM in a NUMA context. 64

Running a thread Running a thread refers to allocating CPU time to a thread that is ready to run. When representing the scheduler as a queue of threads, running is the act of popping a thread from the front of the queue and putting it onto a processor. The thread can then accomplish some or all of the work it is programmed to do.

Synonyms : None. 101

Scheduling a thread Scheduling a thread refers to notifying the scheduler that a thread is ready to run. When representing the scheduler as a queue of threads, scheduling is the act of pushing a thread onto the end of the queue. This operation does not necessarily mean the thread is guaranteed CPU time (Running a thread), *e.g.*, if the program terminates abruptly, scheduled threads never run.

Synonyms : Unparking. 50, 58, 65, 68, 69, 71, 101

System Load The system load refers to the rate at which threads are scheduled versus the rate at which they are run. When threads are being scheduled faster than they are run, the system is considered *overloaded*. When threads are being run faster than they are scheduled, the system is considered *underloaded*. Correspondingly, if both rates are equal, the system is considered *loaded*. Note the system is considered loaded only if the rate at which threads are scheduled/run is non-zero, otherwise the system is empty, *i.e.*, it has no load.

Synonyms : CPU Load, System Load. 11, 20, 77

Thread A thread is an independent sequential execution path through a program. Each thread is scheduled for execution separately and independently from other threads. Systems offer one or more concrete implementations of this concept, *e.g.*, kernel-level thread, job, task. However, most of the concepts of scheduling are independent of the particular implementations of the thread representation. For this reason, this document uses the term thread to mean any of these representation that meets the general definition.

Synonyms : Tasks, Jobs, Blocks. 3, 4, 7–13, 15–17, 20–29, 32, 33, 36, 37, 40–43, 48–53, 57–69, 71–74, 76, 77, 82, 83, 91, 92, 100, 101

User-Level Threading Threading model where a scheduler runs in users space and maps threads managed and created inside the user-space onto kernel-level threads.

Synonyms : User threads, Lightweight threads, Green threads, Virtual threads, Tasks. 2, 5, 6, 15, 17, 33, 56, 78