UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Research project report title page

Candidate 2094I

"Using effect handlers for efficient parallel scheduling"

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 67

Main chapters (excluding front-matter, references and appendix): 46 pages (pp 7–52)

Main chapters word count: 14872

Methodology used to generate that word count:

```
   gs -q -dSAFER -sDEVICE=txtwrite -o - \
   -dFirstPage=7 -dLastPage=52 report-submission.pdf | \
egrep '[A-Za-z]{3}' | wc -w
14885
```

Abstract

During the last twenty years, the advances in computer processors have taken a much different route than before. The exponential growth of frequency stagnated due to physical limits, and CPU manufacturers started to scale processors out, i.e., stack more and more cores within a single processor. In turn, programs became more concurrent to take advantage of the available parallelism, but also because modern computers are more interactive, networked, and omnipresent.

In this dissertation, we argue that modern hardware is so parallel and workloads are so concurrent that there is no single, perfect scheduling strategy across a complex application software stack. Therefore, significant performance advantages can be gained from customizing and composing schedulers.

To validate the hypothesis, we have developed several practical userspace schedulers and extended them with a number of work distribution methods. The code was written in OCaml with multicore support, which features a novel effects-based approach to multithreading. Most importantly, it decouples lightweight threading from the runtime and lets user compose schedulers.

The evaluation involved several real-world benchmarks executed on up to 120 threads of a dual-socket machine with two AMD EPYC 7702 processors.

The results show that scaling applications to high core counts is non-trivial, and some classic methods such as work stealing do not provide optimal performance. Secondly, different scheduling policies have a profound impact on the throughput and latency of specific benchmarks, which justifies the need to compose schedulers for heterogeneous workloads. Further, a composition of schedulers in a staged architecture was shown to provide better tail latency than its components. Moreover, the performance of the scheduler developed in this project was shown to improve over the existing default Multicore OCaml scheduler - Domainslib. Finally, the results put in question a common design of overflow queue present in e.g., Go and Tokio (Rust).

Contents

Chapter 1

Introduction

This section introduces and motivates the central problem of this dissertation in §1.1, and provides an overview of the findings in §1.3.

1.1 Thesis

Twenty years ago, computer processors looked very different than today. The first stock non-embedded multicore processor has just been released [57], but customers would have to wait until 2004 to see the first multicore x86 CPU - dual-core AMD Opteron [2]. Up to four could have been stacked inside HP ProLiant DL585 server. At the time, a revolutionary technology that allowed a further exponential increase of computational power in chips [26] in the light of the slowdown of Dennard scaling [16]. This technological direction is still relevant today and remains a large driver of performance improvements. Today, a standard dual-socket machine with two stock processors (AMD EPYC 7702 [1]) offers 256 logical cores. In the meantime, the maximum clock speed between these two processors improved by only 20% (respectively 2.8 GHz and 3.35 GHz for Opteron and Threadripper). Thus, software engineers have strong reasons to try to parallelize workloads.

Practice shows that parallel programming is far from trivial. Mistakes are common both in the industry and academia. They happen even in algorithms claimed to be proven [49]. This inability to take advantage of parallelism has a tangible impact on the productivity of IT-using sectors of the economy [38]. Several research spaces are trying to make parallel programming more accessible. One particularly important for modern operating systems and runtimes is scheduling.

A standard scheduling definition explains it in terms of allocating resources to tasks. It captures the essence of the term in a concise and interdisciplinary manner. However, in the context of systems, perhaps, a more illuminating way to think about scheduling is about bridging hardware parallelism and software concurrency. Not long ago, a computer could get by with far less advanced schedulers since hardware offered no parallelism. Likewise, workloads used to be less concurrent and less networked. Programs at the time had simpler interfaces than nowadays, and overall fewer programs ran concurrently on a single computer. All that made interactivity and fairness a far more minor concern than today.

This simplicity is no longer the case with processors scaling mainly horizontally and ever more complex applications. Thus, this dissertation argues the following thesis.

> Modern hardware is so parallel and workloads are so concurrent that there is no single, perfect scheduling strategy across a complex application software stack. Therefore, significant performance advantages can be gained from customizing and composing schedulers.

The first part of the statement is akin to the construction of a No Free Lunch theorem [31] for schedulers. While there is a growing body of NFL theorems for optimization problems (e.g., [72], [33]), such fundamental results formalize knowledge rather than produce direct guidance for real-world engineering. Therefore, this dissertation aims for a different exposition of the problem, one that analyses practical differences between scheduling policies. It considers widely-used performance metrics and explores the trade-offs between pragmatic scheduling schemes guided by benchmarks executed on stock hardware.

The following sections strengthen the thesis from the view of hardware parallelism (§1.1.1), software concurrency (§1.1.2), the opportunity presented by OCaml Multicore (§1.2.2), and concludes with an overview of the findings in (§1.3).

1.1.1 Parallelism

Moore's law is an observation stating that the number of transistors in integrated circuits tends to double every two years [48]. Around the turn of the millennium, increasing single-core performance at an exponential rate was no longer possible due to physical limits. Thus, manufacturers started stacking multiple processing units within a single chip [28]. As shown by figure 1.1, the exponential growth of transistors continued with the exponential increase in cores. While some researchers conjecture this trend to slow down [25], [26], we still see rapid, if not exponential, growth in this area.

Hence, parallelism is here to stay. Naturally, in contrast with clock frequency increases, schedulers must be carefully crafted to take full advantage of the horizontal scaling of the underlying architecture. That is because:

- Synchronization primitives like locks or atomics do not scale endlessly. A naive work stealing scheduler may have been good enough on a 16-thread Intel Xeon in 2012 but fails to utilize all 128 threads of a contemporary AMD ThreadRipper [18].
- Modern high-core architectures feature non-uniform memory. Thus, memory la-

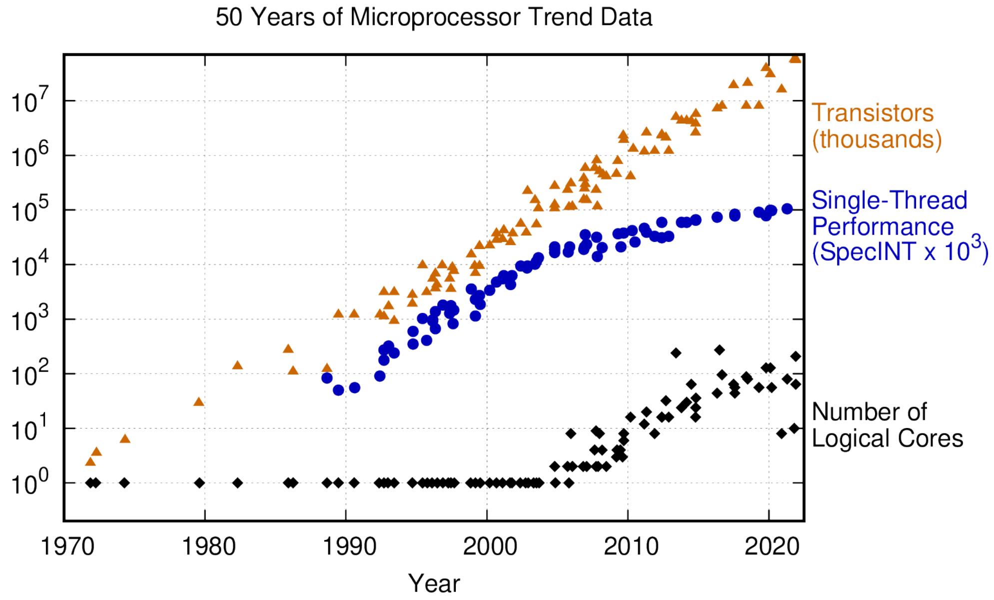

Figure 1.1: 50 years of evolution of processors. The single-core performance sees diminishing returns. During the last 15 years, CPU manufacturers turned to multicore architectures. Figure generated using [53].

tency patterns vary with the topology, and scheduling decisions benefit from taking memory hierarchy into account (e.g., [24], [27]). Moreover, the non-uniformity appears even in consumer products such as Apple M1 or Intel Core i7-1280P, which have two sets of cores: one optimized for performance and another for efficiency.

1.1.2 Concurrency

There are many reason behind why workloads become more concurrent. Firstly, high concurrency is a necessary condition to take advantage of the available parallelism. Compute-heavy applications have strong incentives to distribute work across multiple cores in the world of stalled single-clock speeds.

Secondly, the complexity of the underlying workload increases as computers become more ubiquitous. More useful programs are available with time, and composing their executions requires concurrency. Moreover, the programs alone become more complicated. Data-intensive applications [40] require more concurrency as networked portions of the code benefit from scheduling guarantees.

Finally, concurrency is a convenient tool for expressing specific workloads. Legacy Windows GUI applications are organized around an event loop, which processes inputs from the user. A programmer must carefully judge the handler logic never to starve the loop and cause the UI to freeze. It turns out to be very difficult in practice; for example, a typically small IO may be pointed at a network location and take orders of magnitude longer. A simple compute might stall for far longer on an overloaded system. With multicore support, a modern Go application can spawn a new goroutine for every handler, and rest assured that runtime prevents starvation of the event loop.

1.2 Tools

This section describes the system and programming environment used to verify the thesis.

1.2.1 System

The report aims to investigate runtime scheduling space on a modern, high-core machine. The system features two AMD EPYC 7702 processors, which provide 128 physical cores and 256 threads. It runs Ubuntu 22.04. For more details, see §5.2.1.

1.2.2 OCaml Multicore

OCaml is a unique functional programming language aiming to deliver performance and safety. It found many research and industrial users despite a significant shortcoming for a performance-oriented language - lack of support for multithreading. This confined real-world applications to monadic concurrency libraries such as Async and Lwt, which are troublesome to use:

- **No runtime support.** Lack of runtime support hurts development in several ways, e.g., there is no way to switch between tasks efficiently, and tracing becomes harder as there is no real notion of separate threads and stacks.

- **Blocking.** Combine single-core with no preemption, and any single CPU-intensive or blocking call is prone to cause chaos by starving other threads, e.g., trigger network timeouts. On the surface, there is no qualitative difference with multicore as any n non-preemptive threads can be blocked by n misbehaving task. However, in practice, multicore is far safer. Firstly, a developer may isolate low-latency tasks on a different pool, allowing for more relaxed programming elsewhere. Secondly, a multicore runtime may feature a guardian thread, spawning new OS threads whenever needed [5].

Nonetheless, multicore support has been recently merged into the main tree after a years-long effort. Thus it is an excellent time to take a closer look at lightweight threads schedulers. Significant changes to existing schedulers are notoriously slow and troublesome, as even subtle differences in scheduling policy may expose bugs in existing programs. During my industrial experience, the only way to deal with such a change was to have developers manually verify the code and re-do the testing. Incidentally, both methods are inadequate at finding bugs in multiprocessing. However, this research project will likely contribute to the actual design and implementation of the OCaml scheduler.

Finally, Multicore OCaml is a fascinating study object as it includes several novel solutions. Standard programming languages either include threading support, which is tightly coupled with the language itself, or offer no support. Thus, library schedulers cannot offer much beyond simply running scheduled functions in some order. OCaml, on the other hand, features fibers and effects. Together, they allow writing a direct style, stack-switching scheduler as a library. Further, OCaml offers a mechanism to compose schedulers, a much-needed feature for executing diverse workloads whose portions have different optimization criteria [69].

1.3 Findings

Numerous benchmarks show the complex relationship between scheduling policy, workload, available resources, and chosen metrics (throughput, latency). The results support the need for customization (§5.3.1) and composition (§5.4) of schedulers for optimal performance and the need for new methods to handle highly parallel and concurrent workloads (§5.3.1). Finally, the last benchmark shows a significant scalability improvement over the current default scheduling library in OCaml (§5.6). See section §6 for more details.

Chapter 2

Background

This section begins with a description of the problem of scheduling (§2.1), key concepts, and details of fine-grained multicore programming (§2.2).

2.1 Scheduling

Scheduling attempts to assign resources to tasks in a way that optimizes some metrics [50]. The problem is not limited to computer science and arises in several areas, e.g., organization management. In the case of computer science, it stems from a very fundamental need to compose executions of multiple programs. However, supporting such a model of execution is not trivial. Instructions of multiple programs are inherently impractical to compose since they all write and read the same registers. In effect, composition is achieved by interleaving executions of multiple programs and a special mechanism to hide the existence of these pauses by saving and restoring the state. This leaves us with a clear-cut scheduling problem as there is a need for a holistic decision-making process to determine which program runs when and for how long. Figure 2.1 illustrates this decision-making perspective. Such a model of the problem will be helpful to keep in mind throughout the rest of the report.

2.1.1 System threads

System threads are a fundamental OS feature that detaches the number of synchronous executions from the number of processing units. All kinds of programs grew to rely on this abstraction heavily. For example, Google Chrome spawns a single process per tab [3] and each such process may spawn multiple additional threads. This aggressive acquisition of resources leads to the smooth operation of the web browser. However, it puts significant pressure on the operating system, which has to ensure that in the sea of threads, everyone gets a fair share of the CPU time and the individual objectives are achieved, i.e., ones of a heavy web browser and a latency-sensitive audio driver.

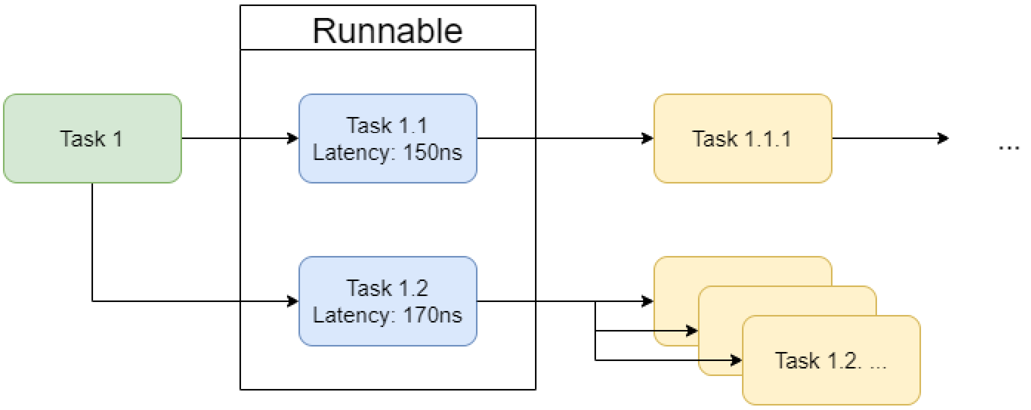

Figure 2.1: A simplified graph of fine-grained tasks, which a scheduler attempts to traverse in order to terminate. The green box represents executed tasks. Blue boxes are currently enqueued tasks. The orange boxes are tasks that will be scheduled once their parents execute, and they are not visible to the scheduler at the moment. This graph shows how little is known about the tasks, and scheduler traps in the shape of infinite-depth subgraphs are prone to cause starvation.

2.1.2 Lightweight threads

System threading provides the fundamental mechanisms to execute concurrent workloads across parallel hardware. Nevertheless, programmers must be wary of designed computations as the cost of interacting with the kernel is non-trivial. It may vary from mode switching to syscall overhead, which requires a number of things to happen (e.g., context switch, capability check, audit). This overhead thwarts the cost of a small task. Thus, the performance of highly concurrent workloads suffers significantly.

These overheads are inconvenient for the developers. Spawning little concurrent tasks is the natural way to express many typical workloads, e.g., processing HTTP requests. In turn, language designers developed lightweight threads, a threading solution implemented in the user space. Lightweight threads aim to be cheap to use, and they do deliver on that promise. Context switch in Go requires saving and restoring three registers [6]. In OCaml, just two [56]. Conversely, a system scheduler needs to save and restore *all* registers and perform mode switching [6]. Hence, lightweight threads are naturally better suited to support fine-grained concurrency.

2.1.3 Preemption

Preemption is the ability to suspend other tasks without any cooperation on their side. It is a cornerstone of modern OS-level scheduling, wherein each program is assigned a quantum of CPU time. Once the time is up, the scheduler preempts (forcibly deschedules) the offender. Preemption is necessary to keep the system stable in a world with imperfect programs. In practice, preemption has more profound consequences. Individual programs no longer worry about their composition with others. They may cooperate, but ultimately, all decisions are made by the privileged scheduler, which has a view of the whole system.

Unfortunately, true preemption is not possible within the user space. Operating systems rely on hardware support for privileged mode and interrupts. For security reasons, a standard application should not be able to access that. Furthermore, even if there was a safe mechanism to allow true preemption in user space, such a mechanism might defeat its purpose. Lightweight threads aim to be cheap, which conflicts with using OS-level context-switching mechanisms.

2.1.4 Work stealing

Work-stealing is a widespread concept applicable to all kinds of multicore scheduling. It was introduced at least 40 years ago [14] and remains used in some of the most cutting edge schedulers, e.g., Completely Fair Scheduler (the default scheduler in Linux) [7], Cilk [15], Go [5].

Work-stealing splits the global work queue (in the FIFO case) into many local, per-CPU runqueues. Each processor strongly prioritizes its runqueue; it will enqueue and dequeue elements from its own structure as long as possible. Now, if the local dequeue is impossible (due to lack of items), the processor chooses a victim randomly and steals some of its work, thus balancing the load. Many benefits flow from such an organization.

- **Less synchronization and faster hot path**. Since local runqueues are single-producer, the enqueue operation needs little synchronization. Likewise, local dequeue operation only needs to synchronize with stealers making it fast as well.
- **Low contention**. Whenever load on the system is high, processors converge to using their queues, thus eliminating any need to compete with others for work.
- **Locality**. Children's tasks are likely to execute on the same processor as their parents, maximizing the locality and improving overall performance.

Chase-Lev double-ended queue (deque) is one of the notable, classic work stealing implementations. Its design is similar to the above example, but while steal remains FIFO, local insertions and removal follow the LIFO order. It minimizes required synchronization since the local thread operates at one end of the structure and steals at the other. Such a setup also improves the locality of the workload 4.3.

2.2 Multiprocessing

Multiprocessing is crucial to taking advantage of modern processors. There are several multicore programming paradigms, each providing different trade-offs between safety and performance. While it is rarely the case in software engineering practice, schedulers are critical enough to choose lock-free programming.

This section introduces vital multiprocessing concepts with a focus on the lock-free perspective.

2.2.1 Amdahl's law

As already discussed, taking advantage of parallel hardware requires extra engineering effort. The actual amount of effort varies wildly between tasks. In some cases, parallelization is trivial and boils down to using relevant threading API, e.g., computation of five independent hashes.

However, on the other end of the spectrum lie programs for which effective parallelization is impossible. Assume computation of a chain of 5 hashes, wherein each hash depends on the output of the previous one. The hashing operations need to happen sequentially. There might still be an opportunity for parallelization, but only within a single hash computation. Even if hash operation is very parallelizable, the synchronisation will need to happen before and after every hash operation in the chain (10x), rather than just at the beginning and end of the workload.

Amdahl's law describes the speedup coming from parallelization [30].

$$Speedup = \frac{1}{1-p+\frac{p}{n}}$$

Where n is the number of processors and p portion of the program that is parallelizable. The law puts a strict bound on the gains from multiprocessing. It also hints at why adding more cores may yield diminishing returns; doubling the number of processing units halves the time spent in the parallelizable portion of the workload. Thus, gains on parallelizable portions rapidly become smaller than any non-trivial sequential part of the program.

2.2.2 Lock-free algorithms

Lock-freedom characterizes algorithms, which guarantee that at least one of many threads will terminate in a finite number of steps (with locks, if the thread holding a lock is not scheduled, no one is allowed to progress) [30]. Lock-free algorithms use the same low-level primitives that are used to build high-level blocking synchronization, but here they are applied to the problem at hand. Thus, the absence of explicit lock objects does not guarantee lock-freedom, as in theory, blocking behavior can be reimplemented from scratch.

The main advantage of using lower-level primitives, atomic operations, comes from better control over synchronisation, which may translate into better performance. Naturally, performance improvement is not guaranteed; a contentious lock-free algorithm may scale much worse than an alternative implementation based on a high-quality lock.

2.2.3 Atomic operations

Atomic operations are hardware-supported instructions, which guarantee synchronised, indivisible execution. They are mostly non-blocking and operate on at most one processor word. Actual usage and implementations vary between architectures, but languages typically provide a unified interface. So is the case with OCaml. This project relies primarily on two atomic operations:

- **Compare and swap**. CAS takes three arguments, one variable (memory address) and two values: old and new. If the variable equals the old value, it gets updated to the new value in one step. CAS reports success or failure to the caller.
- **Fetch and add**. FAD takes two arguments, one variable (memory address) and an integer value. The integer is added to the variable, and the prior value is returned. FAD always succeeds.

Both operations have similar latency when applied to the same use case [54]. CAS has the benefit of being much more general but at the cost of potentially having to retry. It is worth pointing out that these operations essentially push the burden of synchronisation to the microarchitectural level, but it remains non-trivial. Atomic operations may still take one or two orders of magnitude more cycles than their non-atomic counterparts and most common instructions.

Implementations of atomic operations may allow or forbid particular kinds of memory reordering [8]. OCaml atomics only come with sequential consistency, which forbids any reordering across an atomic operation. Therefore, this report does not consider memory order relaxations.

2.2.4 Linearizability

Linearizability is a property of execution trace, which states that all actions can be recomposed into a correct, sequential history. It is crucial whenever operations execute concurrently. High-level concurrency primitives such as locks help to achieve this property trivially; if a single lock protects the relevant state, then any functions linearize since they are forced into sequential order.

However, when performance is the priority, a common approach is to distribute the state to allow many agents to act in parallel. Here, linearizability is easy to give up involuntarily, and it becomes a critical property to watch. For example, the design of a queue does not generally prohibit enqueuer and dequeuer from operating independently on queue's different ends. Such behavior can be achieved using atomic operations to modify *head* and *tail* indices, which effectively doubles the throughput offered by a lock. However, suddenly a simple size function *tail - head* may produce gibberish output as it no longer linearizes.

2.2.5 DSCheck

Writing concurrent programs is an error-prone process. This is true for all applications, but in particular, if they use the lowest level synchronisation primitives, atomic operations. Humans are not good at the precise reasoning required to deal with super-exponentially increasing interleavings, which occurs as more functionalities are added, and the state becomes fragmented. Generally, there is no solution to this difficulty. Commonly, academics attempt to write formal proofs for the code, which is also a non-trivial process and has let errors through in the past (e.g., [49] disproves [41]).

CDSChecker [49] is a novel approach to verifying lock-free algorithms. It validates all possible interleavings of a lock-free algorithm, which is extremely helpful in overall validation of the code and tracking down rare race conditions (e.g. occuring a few times in a million executions of tight loop designed to expose such race [12]). Moreover, CDSChecker does not require building any model of the code, which could be incorrect. It simply requires the atomics library to be shadowed in the original implementation.

This dissertation's core lock-free data structures have been validated using CDSChecker's adaptation to OCaml - DSCheck [34].

Chapter 3

Related work

Scheduling is omnipresent in computer science. It may happen at multiple levels simultaneously: data-center orchestration [29], OS-level threading (§2.1.1) and runtime threads (§2.1.2). The field is very rich in literature and applications. Therefore, instead of summarising all the related work (which would be beyond the scope of this report), this section closely looks at the design of two modern yet mature runtime schedulers. Project work and design decisions tie back to this review.

The first analyzed scheduler is part of the Go programming language (§3.1), whose excellent support for concurrency is one of its main selling points. Like OCaml, it is a high-level language with features like garbage collection. However, OCaml strives to be more high-performant and does not feature preemption or a built-in runtime scheduling. Therefore, the second reviewed scheduler is Tokio (§3.2), which is a Rust library. It covers all of the mentioned discrepancies. Finally, OCaml's unique features for supporting concurrency and parallelism are discussed.

3.1 Go

Go is an imperative programming language that aims to be the language for building distributed systems. It draws heavily on Google's extensive experience with building complex back-end applications. Go features deeply integrated support for lightweight threads, called goroutines, whose design has been central to the language since its beginning.

Go standard library adds significant machinery on top of the POSIX thread interface. In contrast with typical OS threads, Go works hard to let developers schedule new routines as freely as possible, allowing hundreds of thousands of them to execute concurrently thanks to, e.g., marginal memory footprint - 2KiB per routine at initialization - and cheap context-switching. Goroutine is flexible in that it can be a short-lived task or a long-running process since the compiler automatically inserts statements returning control to the scheduler in strategic places, like function entry or loops.

On the surface, Go features a classic work stealing scheduler with local queues and steal-half operation to distribute work. There are, however, significant differences. Goroutines are scheduled on a *processor* object rather than a worker (OS thread). That setup decouples runqueues from OS threads, which helps deal with blocking operations. Firstly, if an OS thread becomes blocked, it hands over its entire context (or in other words, has it stolen), ensuring no goroutines linger in its runqueue. Secondly, runqueue-rotation limits the total number of runqueues at the number of logical processors, reducing state fragmentation and maximizing the chance that thieves find work quickly. With such a design, autoscalability emerges naturally as minimizing the number of unoccupied running workers. Notably, runqueues have a constant size of 256 items, and any extra items are pushed into a global locked queue [62].

3.2 Tokio (Rust)

Rust is another modern imperative programming language. It encompasses a much different philosophy than Go. Historically, performance and safety were a trade-off. For example, memory management was either manual (error-prone, zero-overhead) or handled by a garbage collector (safe, non-trivial overheads). Rust strives to achieve both: safety and performance on par with C++. It applies many novel ideas, e.g., the ownership system. They typically come in the form of zero-cost abstractions or static analyzers, as adding runtime overheads would quickly make the goal of matching C++ performance unattainable.

In a similar spirit, threading in Rust's standard library maps to OS threads and leaves building out more advanced management to the user. One particularly successful attempt is Tokio [43]. It is an external library that provides multithreaded runtime and an asynchronous version of the standard library.

Just like Go runtime, Tokio relies on work stealing to share work. However, it does not abstract out OS threads and relies on a more straightforward scheme with a single worker per OS thread and the number of OS threads corresponding to the number of processing units. As the total number of threads is limited and the external library cannot do preemption, such a scheme is prone to liveness issues. Thus, Tokio provides a special spawn_blocking call to spawn new OS threads on demand for blocking or CPU-intensive computations. Tokio also features a global overflow queue if there is no space in the local runqueue. Notably, Tokio relies on a CDSChecker-inspired lock-free validation as well - Loom [64].

3.3 OCaml

As touched on in 1.2.2, OCaml introduces a new flavor of multiprocessing. It brings several benefits, but the most important one from the point of view of this dissertation is the decoupling of scheduling and runtime [58]. In simple words, an external OCaml library developer can now write a powerful, built-in-like runtime scheduler without requiring any changes from the standard library, the runtime system, or the OCaml compiler. This section describes mechanisms enabling this decoupling and other notable features of Multicore OCaml.

3.3.1 Threading

Starting with version 5.0, OCaml offers access to OS threading through the domain interface. System threads are heavy due to the high memory footprint, the overhead of system calls, and context-switching. Domains are similar in that a single domain consists of an OS thread and additional OCaml structures such as a local minor heap [55]. Domains offer coarse-grained parallelism and are optimized for the number of domains not exceeding the number of processing units [65].

To use system threads efficiently, we also need concurrency. Historically, OCaml did not provide any explicit support for writing concurrent programs. Thus, developers using Lwt/Async are forced to divide programs into small functions, which are then scheduled and executed on the same stack.

Multicore OCaml features fibers - light, heap-allocated, resizable stacks. They are non-preemptible. The number of concurrent fibers is limited by memory and may exceed the number of logical cores by orders of magnitude. Furthermore, context-switching between fibers is exceptionally cheap. It requires saving stack and exception pointers of the fiber and loading ones of the target frame [56]. Therefore, they are a perfect foundation for efficient, fine-grained concurrency.

3.3.2 Effects

Effects represent a novel way of modeling not just concurrency but all kinds of computational effects. The concept is a product of research on the theory of computation [52] and, despite sizeable formal underpinnings, it is introduced the easiest by analogy.

An imperative programmer may consider it a generalisation of exceptions [17]; effects allow throwing an exception (performing effect), processing it, and optionally returning to the location of throwing (continuation) to resume execution. For a functional programmer, effects provide similar capabilities to monad transformers but in direct style, without imposing monadic form [56]. Direct style is essential for concurrency, as with Async and Lwt applications end up having almost all code wrapped into the monad to prevent liveness issues.

In the case of threading, effects provide a composable means of stopping and resuming execution. Once effects handlers are set up higher up the stack, any function may perform a yield effect, immediately returning the control to the appropriate handler. The handler receives a *delimited continuation*, which resumes execution of the yielding function on demand. In the case of a scheduler, a handler may put the received continuation into a queue and resume a different task. This logic is implemented efficiently using fibers described in the previous section (§3.3.1). Whenever the effects handler is set up, it instantiates a new fiber for contained logic. When the logic performs an effect, said fiber is unchained from the top of the stack, and a new one is created and attached to execute the handler. Fibers and effects are optimized to support high-performance concurrency.

In the current shape, effects do not mix with preemption. True preemption is limited to OS-level scheduling. In the case of Go 3.1, runtime simulates preemption by having the compiler automatically insert optional yields into strategic places (checkpoints) in the code. A similar solution does not fit OCaml out of the box because it would pollute the entire program with the Yield effect, which defeats the purpose of the effects system. In Go, the runtime and compiler hide this logic from the developer. In the case of OCaml, suddenly, pure, or even no-op functions would emit effects.

3.3.3 Garbage collection

OCaml features a new garbage collector created with multicore in mind. It has two generations: thread-local minor heaps and a shared global major heap. The minor collection is fully parallel. Both heaps collections require a stop-the-world pause [55].

3.3.4 Domainslib

OCaml ecosystem already has a scheduler endorsed by the OCaml team [67] - Domainslib. Since multicore OCaml is novel, Domainslib is not as mature as schedulers in other ecosystems. Its design closely follows classic deque (§2.1.4). Therefore, Domainslib is not dissected as one of the state of the art schedulers. Nonetheless, its performance is compared against this project in (§5.6).

Chapter 4

Design

This chapter describes the design space of a scheduler. First, §4.1 states the desirable properties of scheduling policies and metrics to evaluate them. Afterwards, major approaches are described, and the remainder discusses other design choices (§4.5).

4.1 Objectives

During the early days of computing, the throughput of a system was significantly more important than interactivity. After all, the user interface could be as minimal as punch cards for input and a printer for output. Properties like latency or fairness were in the hands of a human operator. Over time, however, computers became ubiquitous and networked. People and businesses started to depend on automatic processes terminating timely and reliably. Major software companies such as Amazon and Google have established that latency has a tangible impact on their revenues [40].

Therefore, in line with the literature (e.g. [21], [74], [59]), a scheduler should aim to achieve high throughput and some desired latency characteristics. Typically, low tail latency.

- **Throughput**. A scalar value of tasks (or some other unit of work) processed over a time unit.
- **Latency**. The distribution of processing time of tasks. While queueing theory deals with entire distribution, in software engineering practice, it is more common to select one or multiple percentiles as the key metrics. For example, Amazon chose to optimize the 99.9th percentile of latency and deemed higher ones not cost-effective [40]. Nevertheless, a test-suite of a distributed system may consider both the 99.9th percentile and median to understand the performance better. At the same time, a low-latency market maker may focus primarily on the top 20% latency since those are the orders that win the race with their competitors - for everything slower, the actual size of the delay is not important.

Preemptive schedulers may also explicitly prioritize fairness. Naively, the lower latency, the more fair scheduling. However, this does not hold if jobs have different sizes. For example, a scheduler may prefer to starve a rare long-running job rather than keep thousands of small tasks waiting since the latter will have a higher impact on mean or (sufficiently small) nth percentile latencies. However, as OCaml does not feature any preemption mechanisms (§3.3.2) and the developer effectively defines the CPU time, the schedulers developed in this project treat all tasks as drawn from the same distribution. As discussed further in (§4.4.2), staged architecture allows developers to choose between fairness, latency and throughput more explicitly.

4.2 Benchmarks

This section describes the three main benchmarks used to evaluate different scheduler designs on realistic workloads with respect to stated objectives (§4.1). Each benchmark aims to simulate the characteristics of the core of each workload (e.g., branching factors, dependencies between tasks, memory patterns) but at the same time aggressively mock all dependencies such as networking and other IO. Mocking dependencies significantly reduces noise in the benchmarks and improves replicability.

4.2.1 Packet processing

TCP is the backbone of the internet as we know it. It is used by countless applications, from the world wide web to low-latency trading infrastructure. Thus, the algorithms for processing TCP packets became very sophisticated. At any step, they tend only to parse the parts of the packet that are necessary to proceed. Such lazy behavior is critical for performance and security.

Therefore, the first benchmark imitates a TCP server, e.g., a webserver. Firstly, it accepts a packet and spawns a task to scan the packet for newlines, which is the first necessary step to do more selective and possibly parallel parsing afterwards.

4.2.2 Audio mixer

Telecommunication is a relatively low latency endeavor. Research in human-computer interaction generally recommends keeping the latency of (any) response below 100ms for it to feel instantaneous ([46], [13]). Beyond that, users begin to notice the delay, and the experience deteriorates. [11] focuses on mouth-to-ear latency in softphones (software phones) and classifies everything below 150ms as excellent, 150ms-300ms good, and over 300ms as poor.

While 150ms round-trip time may seem like an undemanding deadline for a modern server, a lot can go wrong (and frequently does, as the pandemic experience shows).

- **Network latency**. In practice, the vast majority of the RTT budget is consumed by transport. A packet may have to cross multiple carrier networks or simply a large geographical distance. Overall, the less latency the server imposes, the better the average user experience.
- **Throughput**. Typical VoIP call-leg sends 20ms frames containing a few hundred bytes of audio. A bare-metal FreeSWITCH [44] installation can handle up to 1000 concurrent calls without quality loss. One thousand concurrent call-legs with bidirectional media means processing 100k packets a second. Each with a real deadline much smaller than the mentioned 150ms
- **Audio complexity**. Standard VoIP signaling protocol SIP lets every conference participant negotiate different codecs. Thus, the server must decompress all incoming audio, mix it, compress it, and send it out. In the case of FreeSWITCH, each call-leg has a separate thread for IO and codecs, and another one per conference to mix audio. That results in audio being passed around and buffered a lot, potentially causing contention.

Thus, the second benchmark imitates a VoIP server with many conferences and participants using FreeSWITCH-like [44] audio-processing, audio-mixing thread architecture. This benchmark aims to simulate an enterprise application with extremely complex scheduling patterns and a perhaps not perfectly optimized code, wherein individual threads frequently need to yield to wait for others.

4.2.3 Binomial option pricing

The last benchmark delves into the world of finance. The livelihood of trading shops depends on their ability to accurately and timely price instruments. Pricing typically happens multiple times for a single trade. Before a trade occurs, a valid price must be determined as quickly as possible to capture an existing opportunity, if any. Once the trade confirmation arrives, several other systems react, e.g., risk exposure needs recalculation. It must happen swiftly to halt other trades if necessary, but precision is now critical. Then the whole process may happen again when more information is known, e.g., for accounting purposes.

Options are a fascinating example of a pricing challenge. The instrument offers the owner an option (but not an obligation) to enter a trade at a specified price and date. The binomial option pricing model is a simple and widely used method of computing an option's market value. The model creates a lattice of future prices and backpropagates the expected payoff from every leaf node (akin to dynamic programming). Further, there is a strict trade-off between computational cost and precision, which is controlled by the granularity of the timestep of the computed lattice.

In contrast with other asset classes, computation of option prices is computationally intensive due to the sheer number of instruments. For example, NYSE has 2400 stock listings, which is not a significant number of iterations for a reasonable computation on modern hardware. However, each stock listing has hundreds of options at different strike prices, expiration dates, and contract terms. That constitutes a far less trivial load, especially if the recomputation happens many times a second and there is a need to balance the cost and precision. All this makes this workload a great testing ground for a scheduler.

4.3 Ordering

This section describes the problem of ordering in scheduling.

4.3.1 FIFO and LIFO

First in, first out (FIFO, also FCFS) regime assumes running tasks in the order of arrival. Effectively, that corresponds to traversing the graph in figure 2.1 in a breadth-first manner. FIFO is commonly used in real-world schedulers. As evidenced during the writing of this report, it is exceptionally robust. The mechanism of always taking the oldest element is admiringly effective at preventing high tail latencies and starvation. The latter, arguably, is one of the most undesirable failure modes in a scheduler. Thus, users can use a FIFO scheduler relatively freely and expect no adverse effects beyond a potential performance penalty. Moreover, additional features tend to be easier to implement (e.g., §4.5.3) as this traversal remains robust even with minor reorderings caused by stealing, overflow structures, and incautious workloads.

Last in, first out (LIFO) order takes the opposite approach and always attempts to schedule the most recently added task. It results in a depth-first search traversal of the task-DAG, which hints at its potential for starvation if at least one of the scheduled workloads consists of an infinite graph. The main advantage, however, lies in LIFO scheduling respecting the principle of locality. Tasks tend to be related to each other, and by running the most recently scheduled task, the scheduler maximizes the chance that required data is still in the cache. Since memory is hierarchical, the benefits should be noticeable for several tasks executed in order and for a wide range of memory requirements.

These two fundamental strategies highlight the trade-off between throughput and latency in scheduling (§4.1), which stems directly from the policy on the lowest level: prioritisation of either the oldest task (for latency) or newest (for locality).

Fairness revisited

The previous section (§4.3.1) describes a common understanding of the FIFO and LIFO trade-off. However, as evidenced (§5), choosing LIFO does not need to impact latency negatively. Firstly, if some particular workload benefits significantly from locality, it is

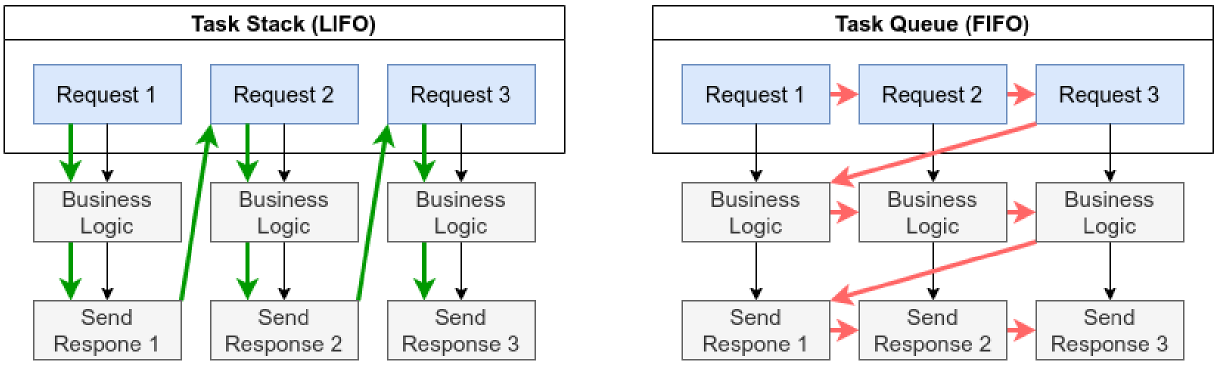

Figure 4.1: The figure shows two ways to traverse the task DAG. Boxes with a blue shade represent tasks physically present in the runqueue (or run stack). The grey boxes constitute tasks that their respective parents will spawn during execution. To simplify, assume that every task is drawn from the same distribution. In such a case, requests processed using FIFO order terminate at timestamps 7, 8, and 9 while using LIFO at 3, 6, and 9. Thus, although FIFO optimizes per-task latency, the latency of actual requests favors LIFO.

likely that both throughput and latency are going to improve. Especially if either stack size is small or tasks move swiftly enough to effectively not spend much time awaiting processing. Secondly, LIFO prioritizes latency measured between subtrees of a workload rather than individual tasks. Naturally, defining unit of work as a task is more practical from the point of view of software engineering. On the other hand, the subtree latency is the better choice in some real-world workloads, e.g., HTTP requests with equal-depth task subtree. Thus, in some real-world workloads, FIFO-order may degrade both key metrics by design. Figure 4.1 shows an example of this phenomenon.

A possible counterpoint to such an example could be to run the entire request as a single task, making the task-latency and subtree-latency overlap. However, the workload does not have to be sequential. It might be better expressed as an actual graph of tasks. Further, splitting the task is the only way to take advantage of parallelism when the system is less loaded. Finally, splitting the request into multiple tasks lets the scheduler make the scheduling decisions. It is desired as the scheduler has the information to make the right call. Moreover, given the sophistication of viable scheduling policies, the application should not need to make such assumptions.

Allocations

Finally, it is worth noting that described effects may be overpowered if a workload is simply more efficient to traverse in either depth-first (LIFO) or breadth-first (FIFO) manner. Such a bias may arise for several reasons. For example, memory, since it does not scale linearly. Hardware is strictly hierarchical, with latency increasing exponentially at every stage. OCaml garbage collector is generational; minor heap is cheaper than major heap (§3.3.3). Sometimes faster scheduling method is simply the one that minimizes alloca-

```
let rec fib n =
  if n < 2
  then 1
  else
    (let a = schedule (fun () -> fib (n-1)) in
    let b = schedule (fun () -> fib (n-2)) in
    await a + await b);;
```

Figure 4.2: Naive Fibonacci implementation.

tions. For example, FIFO traversal of naive Fibonacci implementation quickly exhausts all available memory and experiences a significant slowdown. The same workload under LIFO scheduling causes a trivial load. Naturally, some workloads favor FIFO for the same reasons.

4.3.2 Hybrid

As discussed in the previous section (4.3.1), FIFO and LIFO regimes prioritize fundamentally different metrics. A natural next step is an attempt at amalgamation of the two approaches to identify new points in the trade-off plane and, perhaps, generalize or move towards an adaptable policy. The following heuristics were chosen for evaluation:

- **Stochastic**. Random selection between scheduling the youngest and oldest in the structure.
- **Switch every n**. Toggle between LIFO and FIFO every n tasks.
- **LIFO, reverse stack every n**. It is derived from the observation that LIFO offers superior performance, but tail latency suffers due to elements lingering at the bottom of the stack. Thus, after every n elements, the scheduler reverses the stack, which sacrifices locality in that particular moment to take care of overdue work.
- **LIFO slot**. Have a special slot for the most recently scheduled task. When a new task is scheduled, the item in the slot gets pushed onto the stack. It has the disadvantage that either the item in the slot cannot be stolen without using expensive synchronised operations on all accesses.

4.4 Contention and scalability

Previous sections (§4.1, §4.3) describe global scheduling properties and mechanisms achieving them. However, a simple synchronised queue (or stack) does not scale beyond a few cores. Firstly, imposing thread-safety requires costlier primitives, which are slower even in the single-core case. Secondly, these primitives have the same throughput regardless of the number of OS threads executing them. In fact, with many threads trying at once, their performance may slow down due to, e.g., more CAS retries or microarchitectural effects. Thus, with such a simple structure, the actual parallelizable work must be large for Amdahl's law [36] not to destroy performance gains. To support fine-grained tasks, schedulers tend to take a more decentralised approach with a thread-local data structure for every OS thread. It is described in detail in the following section (§4.4.1).

4.4.1 Core structure distribution

One way to fix the contention is by having multiple instances of the underlying structure and load balancing across them. In the case of FIFO, the multi-queue [51]. Such a distributed structure is no longer strictly FIFO (globally). However, looking at all pairs of processed elements, the number of reorderings from the strict order is small, and enforcement of a hard bound on reorderings is possible [39].

Now, let us focus on multi-queue. There is no need for load balancing on both ends. Instead of having all workers try to insert elements into all the queues and cause contention, each worker could fill its own designated queue. That also improves constant factors because a single-producer queue requires weaker synchronization than a multi-consumer one (§B.5). The load balancing on the dequeue side only maintains the mostly-FIFO property.

There is one more natural optimization in sight. If a worker can take an item from multiple queues, it should prefer to execute the work scheduled by itself. In such a case, it is more likely to benefit from the locality principle and finish faster. Implementing such a bias is trivial, as the worker simply needs to prefer taking items from its insertion queue. Further, such a local deque operation has a lower constant factor because it does not need to synchronize with insertion. Thereby, we have arrived at a more distributed and scalable architecture at the cost of enforcing strict ordering. This design forms the basis of many real-world schedulers, as described in §2.1.4. A similar optimization process applies to LIFO ordering.

To summarize, the sacrifice of the global order property for local, per-core structures aims to decrease contention and increase locality. While any deviation from global order increases tail latencies, in this case, it may be balanced off by an overall increase in throughput and tasks finishing sooner. Naturally, certain formal guarantees have been given up, and that may become more pronounced under specific circumstances, such as during system degradation. The next section discusses an alternative, more stable approach (§4.4.2).

4.4.2 Staged

Staged architecture [71] constitutes a different realization of the FIFO order. Different sets of system threads are assigned to handle particular kinds of logic. Those sets are connected with task queues. For example, a web server may have separate thread pools for parsing packets, managing connection state changes, TLS, and a number for business logic, e.g., reading files and querying DBMS.

Such a model forfeits all the locality between tasks, as progressing to the next stage means being passed to a different system thread. Nonetheless, it offers other benefits.

- **Global state locality**. Tasks are local to the global state of the particular stage. It is beneficial for data-intensive applications (as opposed to compute-intensive [40]) as most I/O operations require the use and modification of a global state at least at some stage. For example, a thread responsible for setting up TCP connection can maintain the list of TCP 4-tuples high in the cache hierarchy. Further, in contrast with, e.g., task-locality in FIFO, locality to global state is less likely to degrade as queue sizes grow.
- **Graceful degradation**. The scheduling architectures with a single structure per CPU described in the previous section (4.4.1) tend to degrade unpredictably. Work-sharing decreases when most threads become overwhelmed and thus focus on local queues for much-needed throughput. That leads to higher variance in task backlogs between threads and, in effect, latencies. Staged architecture performs worse at low and optimal load but degrades more gracefully when overloaded. This stability comes from its simplicity. Overload causes queues to grow, but there are no adverse effects on the execution model to compound the issue [71].

Therefore, even though such a setup is unlikely to provide the top performance for compute-heavy or highly predictable applications, it is a compelling strategy for modern auto-scaling distributed systems. Spinning up new machines always comes with some delay, and it is crucial for service not to degrade before reinforcements arrive.

Finally, staged architecture does not have to be implemented from the first principles. While such an approach would yield the most significant performance benefits for some workloads, it is also the most constraining. Throughout this thesis, staged architecture is instead used as a way of composing different schedulers. Such an approach reaps the main benefits of staged architecture, and tasks may still benefit from, e.g., LIFO scheduling for small tasks within a single pool.

4.5 Other design choices

4.5.1 Work distribution

Work stealing is a widely used realisation of the idea described in the previous section (4.4). It assumes a thread focusing on the tasks in its local structure for as long as it has work. Then, it randomly selects another worker and steals some of its items. Work stealing has proven to be an effective and robust approach in the past. Nevertheless, this report argues that work stealing alone is insufficient for the present and future (§5.3.1).

Despite its simplicity, work stealing can be assumed to serve two distinct roles in non-preemptive scheduling. Firstly, if only some cores spawn the work (e.g., a single web server), stealing helps distribute the work among many cores. Secondly, if a task unexpectedly occupies the CPU for longer than expected, it helps to reduce the latency of tasks in its local structure. In theory, the developer can prevent the latter. However, in my experience with Async, it still happens frequently as the system evolves. Even assuming perfect code, once in a while, an IO operation or database query hits the scalability ceiling and suddenly becomes much slower (*falls of a cliff*). It could be due to suddenly not fitting into some level of memory hierarchy or a DBMS changing optimization plan as more data is processed. Multicore does not solve the problem of CPU-hogging tasks but offers tools to alleviate the issue.

The latency reduction case is difficult to improve upon. A thread cannot know when it will get blocked by a lengthy operation. Therefore, the unoccupied threads must actively search for lingering work, and *stealing* needs to be non-cooperative. (Alternatively, additional bookkeeping could be imposed on every thread but incurring a constant penalty is troublesome to justify for a feature, which is most important when the system is not heavily loaded). Thus, a random search conducted by work stealing is a plausible mechanism to soften the impact of extensive blocking on latency. If predictable latency is necessary, the staged architecture might be a better choice (§4.4.2).

On the other hand, this mechanism becomes harder to accept for regular work distribution cases. Any increase in threads is evened out by lowered chance of individuals finding work. Thus, a software engineer needs to be increasingly cautious to achieve high utilization. Notably, in this case, the information is available and communication possible. The high-spawner can issue a call for help or offload accumulated tasks into a shared data structure. Thus, several work distribution extensions and improvements have been developed and evaluated (§5.3.1).

4.5.2 Resizing

Having a runtime scheduler allows users to express workloads concurrent far beyond what is reasonable with pure system threads. It leads to some algorithms creating large numbers of tasks, large enough to fill up their local data structures rapidly. Now, a common strategy is to simply send new tasks to a slower, global overflow queue (§3.1, §3.2).

This report argues that a global overflow ultimately bottlenecks task creation as the creator is forced to do offloading. Ultimately, if the workload is imbalanced and there are other threads ready to help, high-spawner should focus entirely on the creation and leave the distribution to others. Therefore, a different strategy has been tested: resizing of local structure. Here, the spawner increases its local structure to a size that lets others steal work effectively. This change takes the burden of transfer off the spawner and makes the process quicker as it does not have to contend or even synchronize with any other writers on its local structure.

Notably, the local structure can scale either vertically or horizontally. The latter has the benefit of decreasing contention between stealers.

4.5.3 Yield

Yield operation allows fibers to request descheduling. Commonly, a well-behaved program is expected to call yield to leave CPU while awaiting an event. That avoids wasting resources and blocking other threads which can make progress. However, if the awaited event is a result of a computation done by another thread, yield is necessary to allow both threads to terminate. Thus, it becomes a vital part of the logic. Thus, the design decisions behind this small part of the mechanism profoundly affect the entire scheduler's usefulness. As described below, they are also not trivial.

Firstly, a pure LIFO-based scheduler is not practical as the underlying data structure prevents the implementation of yield operation. LIFO can only insert at the top of the stack, and thus yielded task is the only one that can be picked up afterwards (other than switching to costly self-stealing). Thus, LIFO ordering needs to be relaxable when needed. Heuristics such as first popping the element and then pushing yielded task tend not to fix the underlying issue but increase the number of blocked threads required to deadlock. A more elaborate approach swaps the yielded task with a random item in the stack and runs it instead. That fixes the underlying issues, but overall performance deteriorates if most tasks in the stack are blocked. Also, adding a swap operation is not possible for all lock-free stack implementations. Finally, with naive yield, even FIFO work stealing scheduler may waste cycles or deadlock. Work stealing mechanisms tend to bias strongly towards using local, per-processor structures and stealing only if necessary. Such a construction improves workload locality, limits contention and superfluous passing of tasks between processors. However, with a naive yield, the thread prefers to keep spinning on a single yielding task rather than take more work from peers.

Therefore, the final implementation puts yielded tasks into a completely separate, global queue, which workers check periodically. This approach avoids described issues at the cost of a small extra cost in handling yielded tasks, which is acceptable because yielded tasks are unlikely to be latency sensitive or constitute the majority of the workload. If that is not the case, either the scheduler can decide whether a slow or fast path should be used or, perhaps, more than one yielding mechanism should be exposed - one for fast yields (*soft yield*) and one to be called every-n yields to break the deadlocks.

Notably, Linux kernel developers made a choice in a similar vein in 2003 when yielded tasks were changed to enqueue to expired queue rather than run queue [23].

```
val await : 'a Promise.t -> 'a
val schedule : (unit -> 'a) -> 'a Promise.t
```

Figure 4.3: Implemented promise interface.

4.5.4 Promise

Promise is a placeholder value returned by a scheduler for a newly added task. It is similar to an option type, which is automatically filled with returned value upon completion of respective task. If another thread attempts to read the promise when empty, it becomes blocked until the result is known [61].

Fundamentally, promises do not have to be implemented in the runtime. They are a convenient abstraction to avoid busy waiting and can be built entirely from standard synchronisation primitives. It is the case in Go [20]. With preemption, goroutines tend to be more long-lived than tasks in Tokio or OCaml (and channels offer similar semantics if required). Nevertheless, this project has included promises as OCaml tasks should be very granular. Availability of promises allowed audio mixer (§4.2.2) benchmark to be written in a more idiomatic way. Figure 4.3 shows the implemented interface. Naturally, it is an eager mechanism - scheduled tasks are executed even if nothing awaits the promise.

Chapter 5

Implementations and evaluation

This chapter begins with an overview of implemented schedulers (§5.1) and the principles behind them (§5.1.1). The next section describes methods (§5.2) and what follows is a detailed analysis of evidence gathered to verify the hypothesis (§1.1).

First, it shows that some classic ideas like work stealing with an overflow queue (used in Go and Rust) do not scale to modern hardware and concurrency (§5.3.1). This result is analyzed in-depth (§5.3.1), and a number of extensions favoring fine-grained concurrency are discussed (§5.3.1) and evaluated (§5.3.1). Further, the section focuses on more complex workloads and studies emerging trade-offs between schedulers: web-server with cache effects (§5.3.2), bimodal task processing time (§5.3.3) and LIFO heuristics (§5.3.3), mixer benchmark (§5.4), and binomial option pricer (§5.5).

5.1 Schedulers

As explained in §4.3.1, there is an inherent trade-off between FIFO and LIFO scheduling. These two mechanisms have been implemented from scratch as baseline choices. Their design follows the scheme with local data structures (§4.4.1) and relies on work stealing for work distribution (§4.5.1). They are built from first principles and supported by high-performance lock-free structures (§5.1.1).

As shown by §5.3.3, §5.3.1 the fundamental FIFO and LIFO schedulers can be limiting or outright fall short of expectation. Thus a large number of alternatives, modifications, and extensions have been tested. They can be divided into 3 groups:

- **Staged architecture.** The staged architecture utilizes the baseline schedulers as micropools and provides scalable means for scheduling tasks across them (§4.4.2). This implementation uses a scalable, lock-free multiqueue (§B.8).
- **Resizable FIFO.** Work stealing FIFO scheduler, which grows local queues to match the load rather than push items into an overflow queue. It uses the same lock-free

queue as FIFO scheduler extended with resizing mechanism (§B.7).

- **LIFO heuristics.** LIFO order may produce poor tail latency for workloads with deep task graphs. §5.3.3 evaluates a number of heuristics that try to capture the benefits of LIFO while avoiding the edge case by doing FIFO removal at strategic times. They are implemented as a wrapper around the LIFO scheduler.

- **Work distribution extensions.** Work stealing is a classic work distribution method, which has some disadvantages. Several work distribution extensions have been considered. §5.3.1 discusses both the disadvantages and proposed extensions.

5.1.1 Design principles

Lock-free data structures tend to be challenging to use in the real world. Firstly, all assumptions about their naive counterparts are off once fine-grained concurrency is in place. For example, a simple size() method may turn out to be non-trivial to make linearizable (§2.2.4) when the size of a circular queue changes in multiple places at once. This phenomenon makes the trade-off space far more complex and points in that space (algorithms) more varied than in the single-threaded world.

Secondly, the effort required to add new features increases as a function of current complexity. With a single-threaded code, the developer implementing a new function can focus solely on the state before and after its execution. That alone is enough to examine whether the new function fits the rest of the codebase. Moreover, if the code has no side effects and illegal states are not representable, even just a type check provides high confidence. Such a function is correct as long as its input and output pairs follow requirements. These principles could not be further from the truth with lock-free concurrency, wherein every new function must be safe to execute concurrently with the whole existing codebase. In fact, even if the new function is correct, it may render seemingly unrelated portions of existing code non-linearizable.

Therefore, this thesis relies on lock-free structures inspired by Kappes and Anastasiadis [35]. Their designs are described in detail in §B. In general, they are optimized for the following properties rather than constant factors.

- **Scalability**. Prefer to use fetch and add operation (FAD) for modifying indices. CAS and FAD operations take approximately the same time [54]. However, traditional CAS-based queues scale poorly due to the number of CAS retries growing proportionally with the number of threads. Here, (despite additional item-level CAS), the contention is resolved by FAD, and the whole structure scales similarly to state-of-the-art FAD queues (e.g., [73]).

- **Flexibility**. Lock-free implementations can be unwieldy and tricky to debug. It posed a particular threat to this dissertation. The described queue is extremely flexible for a lock-free algorithm and allowed exploration of many modifications at relative ease, e.g., LIFO pop operation, non-blocking returns, and single-producer relaxations. Extra features, e.g. immediate removal of cancelled items, are also feasible (§A.2).

- **Simplicity**. The base case of this structure takes a little over ten lines of code. It is unusually concise for a practical multi-producer multi-consumer LF queue.

5.2 Methods

This section describes the experimental setup, procedures, and statistical methods.

5.2.1 System

The hardware features two AMD EPYC 7702 processors ([10]) plugged into a dual-socket Gigabyte MZ92-FS0-00 motherboard ([68]). It features 1082 GiB of RAM. Caches have the following sizes: L1 64 KiB, L2 512 KiB, and a shared L3 256 MiB. A cache line is 64 bytes wide.

The installed system is Ubuntu 22.04 LTS. All the schedulers and benchmarks in this project have been compiled with OCaml 4.12.0+domains [22], which has a hard limit of 128 domains (OS threads, §3.3.1).

5.2.2 Procedures

Each benchmark has been run 11 times, with the first sample dropped as it tends to be noisier than others due to the lazy loading of libraries and cold caches.

Performance-sensitive data was collected on thread-local counters. Benchmarking latency required many calls to the system clock, which were ensured to be served by vDSO [37] and take less than 50ns on average.

5.2.3 Statistical methods

Since it is hard to assume any particular distribution (esp. gaussian) when it comes to performance measurements, the statistical methods used throughout this project are distribution agnostic. Charts feature medians, 25th, and 75th as error bards.

5.3 Server benchmark

5.3.1 General case

The first benchmark mimics a minimal TCP packet processing scenario. The workload relies on a single spawner, as would with a server listening on IP and port tuple. To minimise the noise benchmark does not use a real network. Instead, the *server* thread

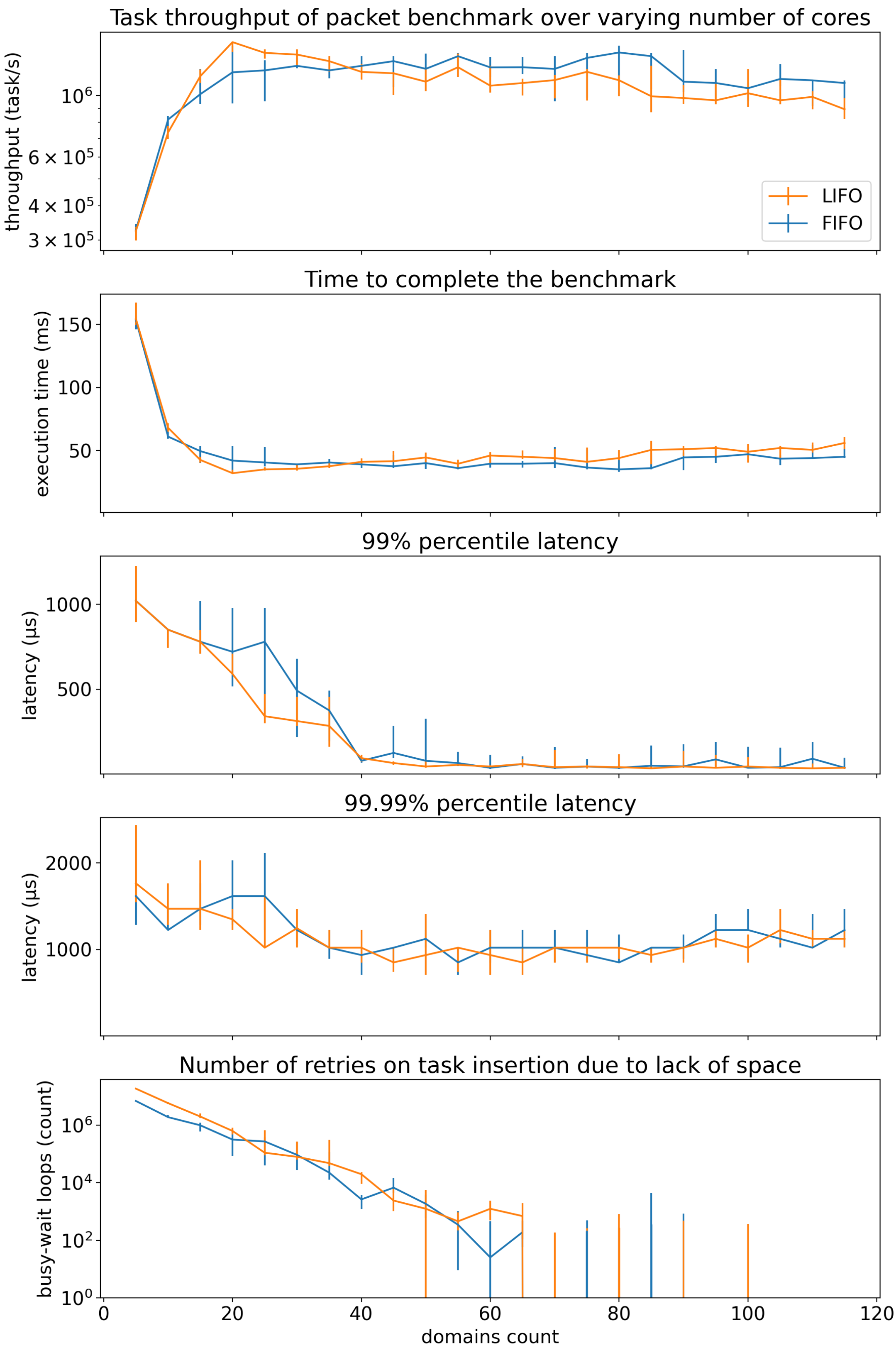

Figure 5.1: The performance achieved by packet processing benchmark with 50k tasks over varying number of cores. The panels show throughput, time, two tail latency percentiles, and retries on insertion due to lack of space.

randomly chooses a pre-made packet, copies it, and schedules a handler task by performing an appropriate effect. The benchmark focuses on the max load scenario and spawns tasks as quickly as possible until the requested total. Afterwards, it monitors an array of thread-local counters, which are increased by successfully terminating tasks. Notably, the packets are between 2 and 3 KiB, and the spawner thread focuses solely on task creation. Therefore, cache locality does not significantly affect the performance. Likewise, the performance advantage of LIFO stealing does not apply to stealing from the spawner, as the spawner never removes elements from its local structure. Local structure sizes are held constant at 128 elements.

The results of conducting a test with 50k packets in such a controlled setting are shown by figure 5.1. The first panel shows the median processing speed on a logarithmic scale. In the first part of the graph, between 5 and 20 domains, the performance rapidly accelerates with the number of threads. Twenty domains witness peak performance by the LIFO scheduler, and the performance degrades as more threads are added. At first, slowly, then rapidly beyond 80 domains. FIFO scheduler sees its peak at 80 domains, but its performance is also relatively stable between 20 and 80, with a pronounced drop afterwards. This is surprising; adding more resources may not translate to a significant improvement [36], but it should not be detrimental either. Discussed further in §5.3.1.

The second panel shows the actual time required to complete the benchmark. Next, the charts from three to five 99%, and 99.99% latency percentiles. Again, up to 20 domains, there is a steep decrease in all latencies. There are local differences around 20 domains, with LIFO offering better tail latencies for 2-3 points. Afterwards, the results converge. The tail latencies are partly driven by LIFO being faster to process tasks but, likely, not as much as it looks since error bars indicate more noisy measurement than elsewhere.

Finally, the last panel shows the number of retries of the main spawner (*listener*) when trying to insert elements into the queue or stack. The retry loop is cheap to execute. The spawner frequently throttles, adding tasks for lower core counts due to lack of space. However, the number of busy-waiting cycles lowers with extra cores, quickly reaching a point where magnitudes are too small to affect overall performance. At last, it ceases to occur in the median case beyond 45 cores.

Both FIFO and LIFO perform very similarly on all metrics and exhibit similar scalability. There are slight differences, i.e., LIFO peaks at 20 domains, while FIFO at 80. Nevertheless, their performances (throughput and latencies) in ranges 0-20, 20-80, 80-115 are comparable. As hinted in the first paragraph, the workload is so simple that it does not let some of the unique advantages prevail. Both implementations are reasonably balanced and constitute a good foundation for benchmarking more complex workloads.

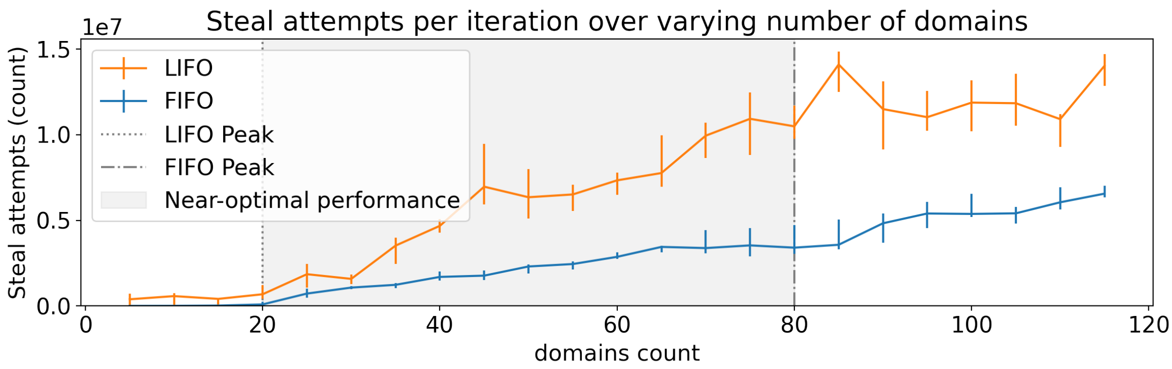

Figure 5.2: The number of steal attempts reached by FIFO and LIFO schedulers on packet processing benchmark with 50k tasks.

Degradation

In line with Amdahl's law [36], it is expected that any application will stop being able to take advantage of more resources at some point. After all, scalability is something to design for explicitly; even then, it may not always be possible. However, there is no justification for more workers causing harm to the performance. Such a characteristic is highly undesirable in a component as foundational as a scheduler. However, why would it arise in such a battle-tested idea as work stealing?

In the LIFO case, as shown by 5.2, the scheduler cannot maintain the increase in stealing attempts. Such an increase is necessary for scalability, as each worker will need more stealing attempts to find the tasks. For example, if 1 out of 2 threads is spawning tasks, the other thread has a 100% hit rate. On the other hand, if it is 1 out of 10 threads, then there will likely be some misses or inefficiencies (minimal amount of work acquired). The diagram shows that above 80 domains, as the information becomes more dispersed, the scheduler does not increase stealing attempts. In theory, there should be no scalability issues since each new worker also adds a new queue. In practice, contention increases significantly as starved cores keep retrying and impeding the actual useful attempts.

In the FIFO case, the problem also relates to work distribution. FIFO is not able to steal as much as LIFO because all steals compete with local dequeues, and, by design, those steals tend to lose the race more often since local dequeue uses a FAD rather than CAS to increment the index (§B.5). Therefore, while stealing alone is not maxed out, it significantly affects the rest of the logic. As shown by figure 5.3, FIFO reaches over 2.5 stalls per instruction beyond 80 domains and this number keeps raising. The effect is also visible for LIFO but its pressure on the entire caching subsystem is much subtler.

Finally, it is worth stating that this result should not be interpreted as work stealing being generally inappropriate for saturating 100 cores. Packet benchmark creates a short burst of 50k items, which reaches a throughput of over 1.5m items per second. Each task has

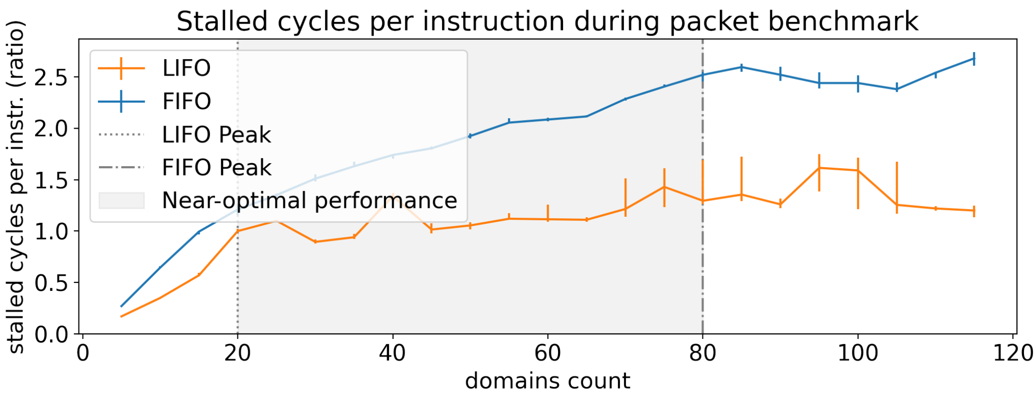

Figure 5.3: The number of stalled cycles reached by FIFO and LIFO schedulers on packet processing benchmark with 50k tasks.

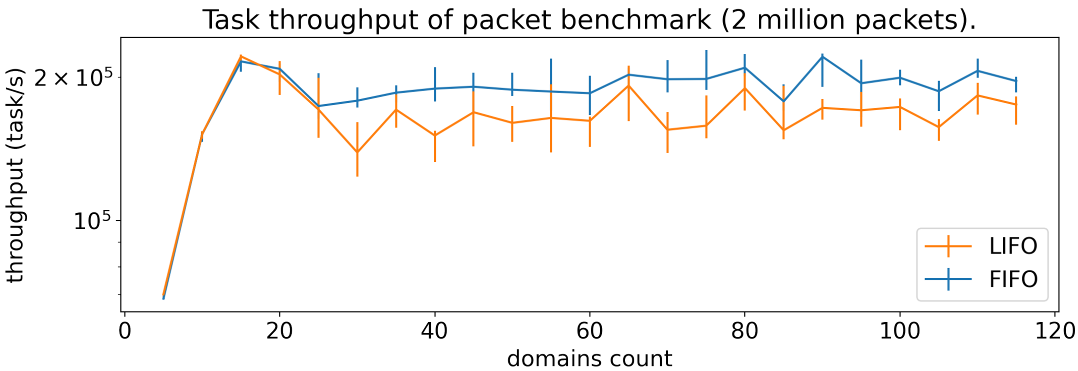

Figure 5.4: The throughput achieved on packet processing benchmark with 10 million tasks.

its logic and requires allocations, but this benchmark does constitute a very fine-grained concurrency. That puts high stress on the scheduler and exposes its inefficiencies. Further, there is an unusual amount of distribution required. It is far easier to distribute tasks onto 10 threads than 100, e.g., atomic operations take constant time, and load information is less dispersed. If the workload is simplified just a little, the scalability improves.

- **Heavier task**. See figure 5.5 for the performance when each task has had an extra 500ns sleep to perform. It is enough time to extend the stability beyond 100 domains.
- **More spawners**. See figure 5.6 for the performance under more spawners, i.e.; the developer made an explicit change to help the scheduler.
- **Higher workload**. Increasing the number of tasks also stabilizes the scaling. See figure 5.4 for the case with 10 million task instead of 50k. This case saturates many more workers, and while overall throughput is lower, the performance remains stable beyond 30 cores.

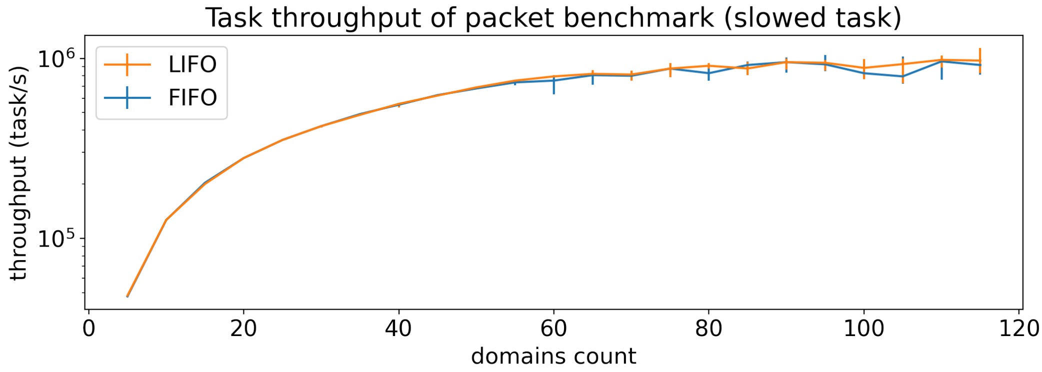

Figure 5.5: The throughput achieved on packet processing benchmark with 50k tasks. Each tasks loaded with additional 500ns wait.

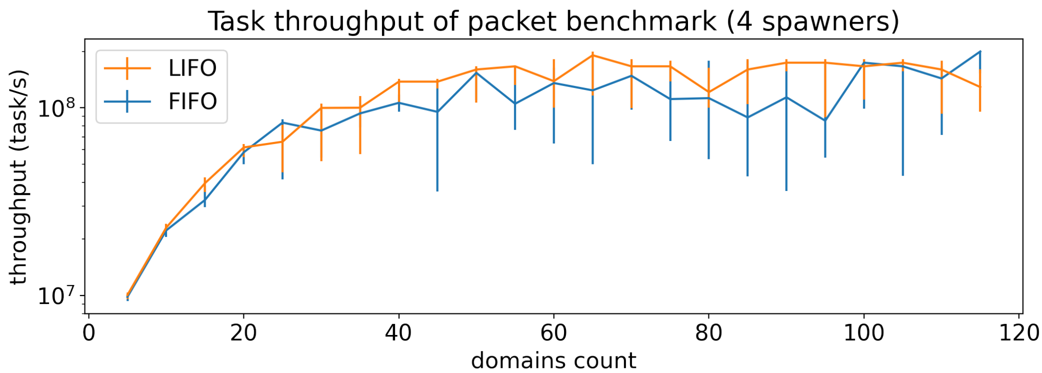

Figure 5.6: The throughput achieved on packet processing benchmark with 50k with 4 spawners. (2 spawners in the 5-core case.)

These differences in scalability highlight the need for new methods to take advantage of high parallelism and fine-grained concurrency (§1.1).

Work Distribution Extensions

Work stealing is robust but not perfect, e.g., the scalability ceiling describes in the §5.3.1. As described in §4.5.1, regardless of the primary work distribution mechanism, some sort of work stealing is necessary to prevent starvation in non-preemptive design with local structure. Otherwise, with purely cooperative scheduling, any long tasks will significantly increase the latency of all awaiting items. Therefore, several extensions to the core work stealing have been considered. They are motivated by the following observations.

- **Contention.** The scalability of atomic operations and inter-processor communication is limited. Therefore, the design of stealing should be wary of those budgets as additional threads are added.
- **Misplanning.** It might not make sense to steal a small number of tasks. If the victim does not have enough work, the thief should simply look for another victim rather than conduct a steal and cause the victim to have to steal soon in effect. A naive implementation of this strategy could lead to starvation on blocking; thus, the implementation has a bias toward stealing a small number of items rarely.
- **Blindness.** Random exploration of other threads' local structures is inefficient. It causes contention and suboptimal steals. Instead, the busiest threads should request help.

The aforementioned observations translated into the following implementations.

- Reduce the contention on critical threads by having each thread only target its n nearest neighbours in the threads array. It creates a hierarchy along which tasks move and prevents the underloaded threads at the bottom of the hierarchy from contending in the hotspots. Implemented with nearest 15 and 20 threads.
- **Steal at least 10.** Do not steal less than ten elements as the owner will process them efficiently. Instead, look for work elsewhere.
- **Work-shedding**. A polar opposite of work stealing. A busy worker randomly chooses another thread and forces it to steal some of the busy thread's tasks. Notably, [45] found an improvement in tail latency stemming from extending work-stealing with work-shedding on a 32-core machine. Implemented as non-sticky, only a single steal is forced and sticky, in which the victim keeps the target until another thread overrides it.
- **Work-sharing from busy to free**. Requestor publishes its id in a highly-scalable lock-free queue. Threads looking for work pick up the id from the queue and steal work. Only the overloaded and underloaded threads participate, while others re-

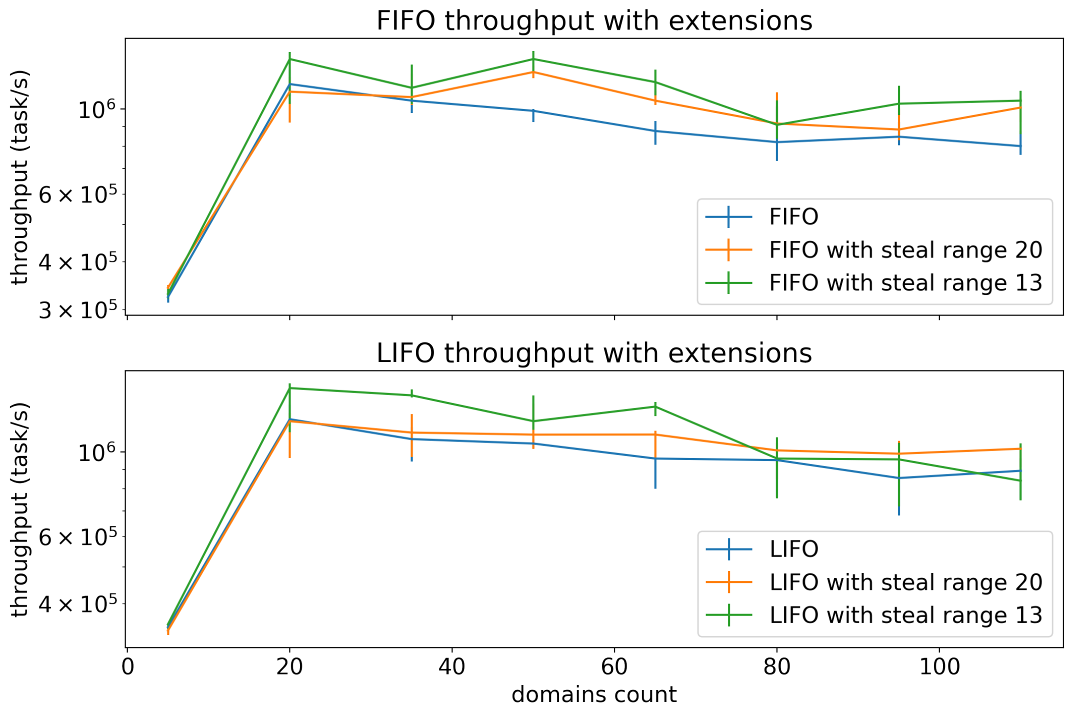

Figure 5.7: The throughput achieved on 50k tasks with over varying number of domains. Each tasks loaded with additional 500ns wait.

main unaffected. It helps close the information gap of pure work stealing or work-shedding.

- **Overflow queues**. Use a different structure for work distribution and put some tasks into a global queue. Here, the mass-spawner thread has a chance to insert tasks into the global structure. Both Tokio [42] and Go [63] feature global, locked queues for any tasks overflowing the local queue. Three queues have been tested: locked, lock-free, lock-free, and 5-multi-queue.

- **Local resizing**. Stealers aim to take half of their victim's tasks. Increasing the size of the local structure and the number of awaiting items may, in turn, lead to a higher number of items distributed per steal.

Work Distribution Extensions Result

§5.3.1 discussed a number of alternative work distribution mechanisms. The positive results are shown in figure 5.7. Only neighbourhood stealing has shown a clear improvement over the baseline, which is in line with the diagnosis stated in the previous section §5.3.1. It is worth pointing out that such optimization will harm coarse-grained workloads, where threads higher in the hierarchy will be stuck processing tasks for longer, thus not replenishing their local structure quickly enough. This result shows that very

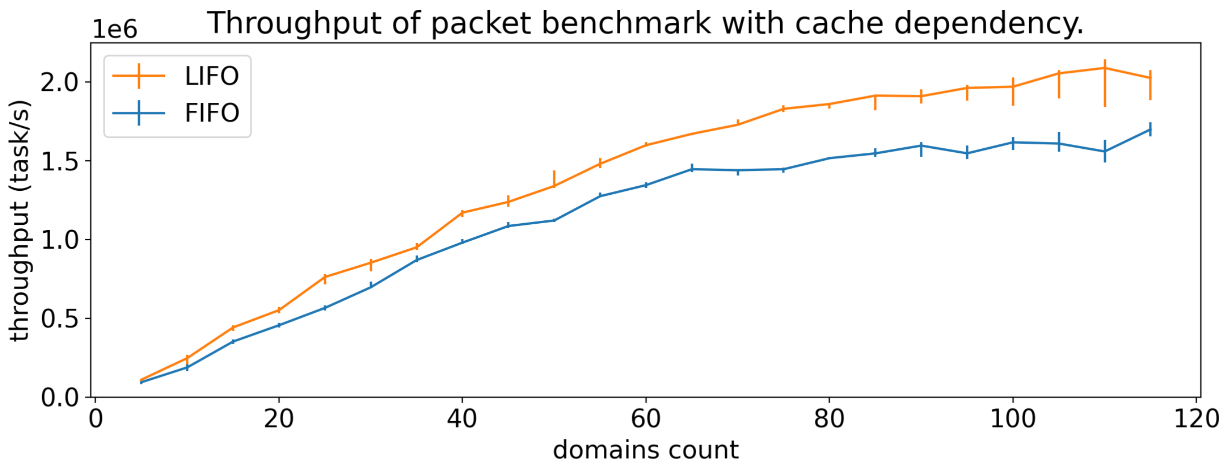

Figure 5.8: The throughput achieved on the packet processing benchmark with 500k packets and ten spawners. The benchmark was modified to have a non-trivial cache effect. Once the task was processed as in the original benchmark, it would spawn a chain of 10 tasks. Each task reads 20, with each reading 20 random bytes from the packet buffer, simulating picking out different parts of the received data at different stages. LIFO offers an obvious throughput advantage (shown in the figure) and median-tail latency improvement.

fine-grained workloads may require novel approaches and support the hypothesis (§1.1).

Only stealing at least ten elements had a strong negative effect, as it paradoxically increased the number of underloaded threads and contention on the loaded ones. Similarly, work shedding (sticky and non-sticky) resulted in other threads overloading the spawner and causing less efficient work distribution than random work-stealing. In pure work stealing case, tasks often flow through multiple hops, thus reducing the contention in the hotspot. The same effect was observed for simple locked overflow queues (as in Go and Rust), as the spawner must compete with the workers for the lock.

Finally, scalable overflow structures (lock-free queue, multi-queue) and work-sharing from busy to free had an insignificant positive effect. They would likely yield a more substantial benefit for a less heterogeneous workload. Likewise, local resizing would be beneficial if starting size of the local structure was small enough.

5.3.2 Server benchmark with cache effects

This section presents the results of a more complicated workload, wherein once the packet is parsed, its contents are passed to the business logic. Namely, each packet from the general case benchmark (§5.3.1) also spawns a 10-sequence of consecutive tasks. Each reads 20 random locations in the packet or does a simple computation (checksum) and spawns its child. That is until the depth of 10 is reached. Moreover, there are five spawners instead of 1, and individual queue sizes have been increased to 1024 to accommodate higher spawning capability. The results are shown by figure 5.8. Such a workload nat-

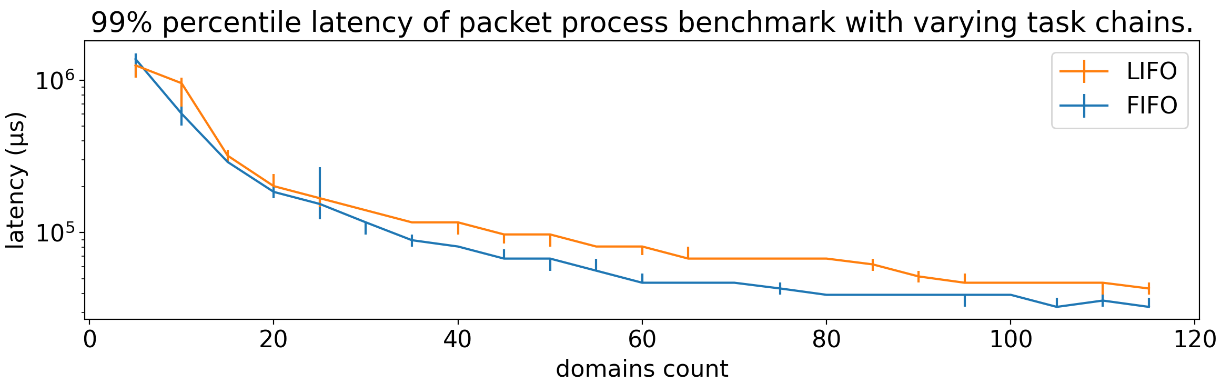

Figure 5.9: The throughput achieved on the packet processing benchmark with 100k packets and ten spawners. The benchmark has a weaker cache effect than 5.8 but is still significant enough for LIFO to have better throughput. However, there is a 0.1% chance that the packet causes heavy work to be done, which spawns two tasks taking 20ms to complete. Akin to a network call to a different service. As shown in the figure, FIFO offers a much better tail latency for most domain counts in such a scenario. For example, at 115 domains, LIFO saves over 30% of tail latency.

urally favors LIFO, which can better use hierarchical memory and reduce cache misses. Thus, achieved throughput is far superior with LIFO ordering. Further, in line with the mechanics described in §4.3.1, using LIFO also significantly improves tail latencies.

Notably, extra care is required to replicate this result on modern architecture. For example, simple iteration over the packet in a child task does not experience significant memory inefficiencies due to automatic memory prefetching. After a few first cache fails, the rest of the packet will be loaded into the cache preemptively. This effect breaks down with more complex memory patterns, which microarchitecture cannot predict.

5.3.3 Server benchmark with bimodal processing latency

Figure 5.9 shows results of a different twist to the packet benchmark than the previous section. Here, instead of 10-sequence, only 3-sequence is spawned, and the last task has a 0.1% chance of spawning another task with a lengthy operation. The 3-sequence still has cache-dependencies that favour LIFO, but the optional final task sets a trap. If the scheduler processes the lengthy operations without care, it risks massively increasing the latency of waiting tasks. This benchmark favours FIFO, which can process lightweight sequences concurrently with the heavy ones, and have the lightweight tasks terminate early. The heavy operations wait longer, but in their case, the processing time is big enough for extra waiting to have a marginal impact.

LIFO heuristics

Pure LIFO scheduling is prone to poor tail latencies. Therefore, a number of heuristics have been implemented. They aim to integrate FIFO and LIFO ordering in a way that

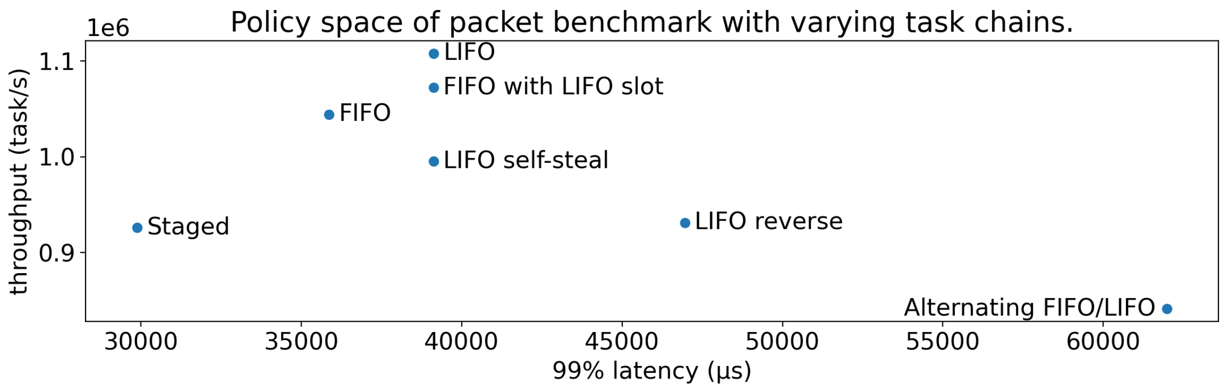

Figure 5.10: The throughput and latency trade-off space achieved by various schedulers on the packet processing benchmark with varied task chains. The frontier consists of pure LIFO, FIFO and staged architecture, whereas various hybrid heuristic offered no benefits.

improves upon both or at least provides a new point in the trade-off space.

- **Alternating**. LIFO and FIFO removal alternately. A simple combination of pro-latency and pro-locality choices.
- **Stochastic self-steal**. Just as alternating, but the ratio does not have to be 50:50. Instead, it has been tested as 80% LIFO, 20 % FIFO, and vice versa.
- **Reverse**. To prevent any items from lingering at the bottom of the stack, every 256 items, do as many FIFO removals as was the number of elements in the stack. Akin to actual stack reversal, but this implementation avoids the edge case of an item getting stuck between two infinitely deep tasks.
- **LIFO slot**. Rely on FIFO ordering but have a special slot for the most recently scheduled task. When looking for work, check this slot first. It violates FIFO and helps preserve locality between a chain of tasks. Featured in real-world projects, e.g. §3.1.

Notably, these heuristic have been built as a wrapper around LIFO data structure but have no direct support. Thus, by design, they are costlier to use.

The baseline approaches described in §5.1, and a number of heuristics described above have been tested. See figure 5.10. The staged architecture was the only one to offer a useful point in the trade-off space, in the form of significantly superior tail latency at the cost of throughput. This gain stems from having a pool of threads dedicated directly to the lightweight tasks. The fall in throughput is difficult to avoid in a workload that work stealing can scale efficiently. Nonetheless, the staged architecture can improve on both metrics if the final task is lighter, as observed with artificial benchmarks.

This result, in line with the hypothesis (§1.1), strongly support the benefits of customizing schedulers. Different policies may be appropriate for the same workload depending on whether throughput or latency is more important. Further, the composition of multiple

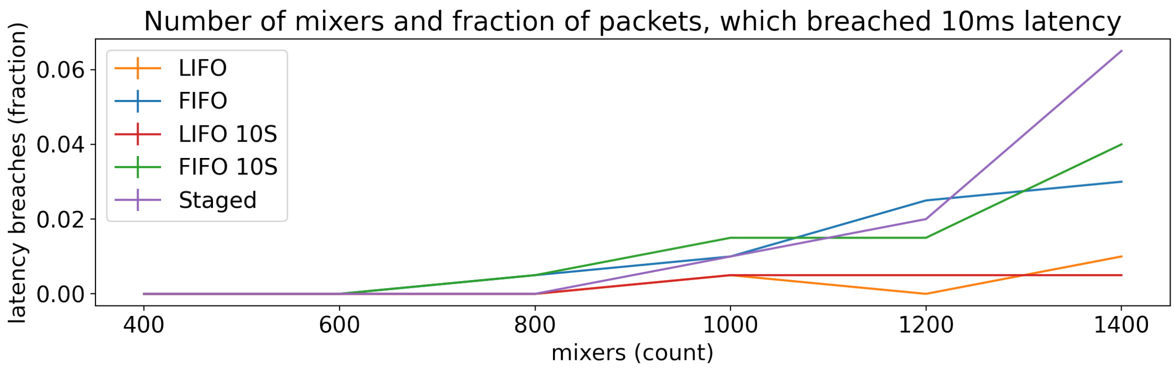

Figure 5.11: The number of latency breaches achieved by different schedulers as the load (number of mixers) increases. FIFO 10S and LIFO 10S represents the schedulers consisting of ten independent FIFO/LIFO silos.

schedulers using queues turned out two offer better latency than both baselines.

5.4 Mixer benchmark

The mixer benchmark represents a complicated real-world use case. It derives from the architecture of a VoIP softswitch - FreeSWITCH ([44]), which heavily relies on threading. FreeSWITCH runs multiple OS threads for a single call. Firstly, there is a thread for the conference. It collects audio from all participants, mixes it (hence mixer), and passes the output to all participants. However, several things need to happen for every participant. At the minimum, each participant can have a different codec setting; thus, the incoming audio has to be decompressed for mixing, and outgoing audio compressed before sending. In practice, there is a lot of other per-call-leg logic as well, e.g., signaling and jitter buffer. Therefore, there is also a single thread per participant. In such a setting, the mixer thread only interacts with participant threads, which interact with the external world.

Such a workload is tough to optimize due to its complexity. It features short-lived and long-lived tasks. Some tasks benefit from cache locality, while others do not. Many details will vary over time, e.g., distribution of the number of people in mixers (as a conference could have anywhere from 3 to hundreds of people), length of calls, codec selection, media routes (IP-to-IP) [70]. Moreover, the workload makes heavy use of many high-level features such as yielding (§4.5.3) and promises (§4.5.4), which are not the typical targets of high-performance optimizations. Therefore, robustness is just as crucial as overall performance in this case.

Importantly, participant and conference threads yield frequently. They simply cannot make progress without data. Yielding threads must be placed in an external global structure (§4.5.3), here a multi-queue. It seems suboptimal because locality is not respected, and centralization of tasks is rarely the desired state. Therefore, one more approach has

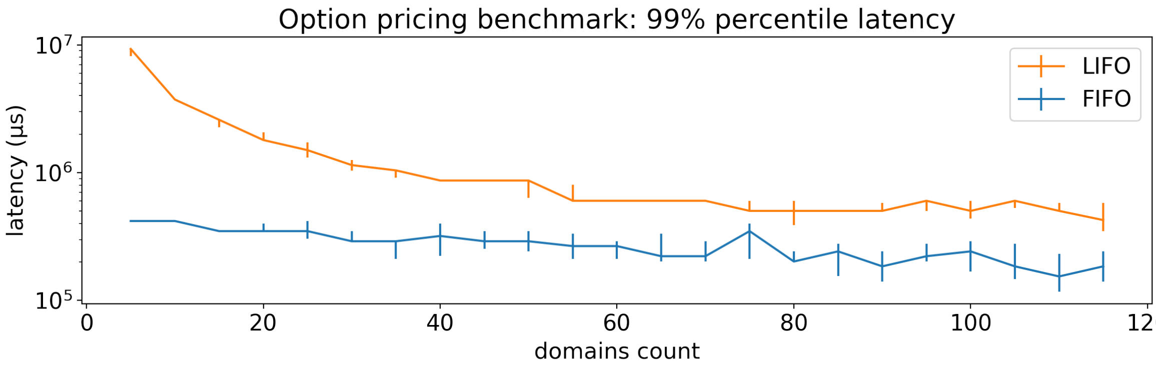

Figure 5.12: The 99% tail latency of achieved by options benchmark under different schedulers.

been evaluated on top of FIFO, LIFO, and staged. It is a composition of 10 individual schedulers (FIFO or LIFO) in order to silo particular sets of conferences and participant threads. Such an organisation imposes boundaries on stealing and yielding since whatever happens, the task remains in its silo. Akin, in spirit, to neighbourhood stealing (§5.3.1).

Figure 5.11 shows the ratio of latency breaches of different schedulers over the increasing number of conferences. This metric behaves similarly to tail latencies. LIFO performs the best as it capitalizes on mixed-down audio being sent back immediately. FIFO sees a significant number of failures earlier, which is explained by the (effectively) constant depth of the task graph (§4.3.1). The staged architecture cannot capitalize on having multiple stages because even in the base FIFO and LIFO case, most of the scheduling goes through the global multi-queue. In this case, the staged architecture only has extra overheads in the form of another queue before sending out mixed-down audio, where it is not beneficial to prioritise other work. Curiously siloed FIFO and LIFO did not differ significantly from core implementations. Locality benefits are not big enough to make a difference in this case. It is also a point in favour of the scalability of the base designs. Notably, all schedulers have been fitted with highly-scalable multi-queue for overflow queues. A scheduler with lock-based overflow structure (as in the case of §3.1, §3.2) does not terminate in reasonable time.

Finally, this result shows that even when optimizing for latency, there are no ready-made solutions. In the case of server benchmark with bimodal packet processing latency (§5.3.3), the best latency was offered by staged architecture and the worst by LIFO. The mixer benchmark is precisely the opposite: LIFO is the best choice for latency and staged the worst. This phenomenon is another example of the hypothesis (§1.1).

5.5 Binomial option pricer benchmark

Options contracts are financial derivative instruments that provide the holder with an option (but not an obligation) to enter a specific trade at some point in the future. For example, a European call option on Google with a strike price of $2600 and expiration date of 2022-03-06 lets the holder buy a specified number of Google shares at the strike price and on the expiration date [32]. The value of an option at expiration depends strictly on the price of its underlying.

$$call\, option\, value = max(underlying\, price - strike\, price, 0).$$

The binomial model first estimates the value of the underlying by assuming that at each time step, the price may either move up by a specific amount or down by the inverse at specific probabilities. It creates a lattice whose leaves represent prices at some final timestep. Then, the binomial pricer calculates payoff at every final timestep and propagates the expected value backward using known probabilities of up and down moves. The first stage - stock prices at final timestep - can be computed analytically, while the expectation propagation must walk the lattice to take interest rates into account. Despite its simplicity, the binomial model is widely used in practice as it is far easier to include extra features than in the analytical approaches.

The benchmark recomputes values of 30k options. Different instruments have varying importance to a trading company due to the size of the current position, available infrastructure, and other models. Thus, most instruments are computed at a reasonably low accuracy (20 timesteps) and 0.5% at very high accuracy (2k steps), which results in a bimodal distribution of work per instrument. The pricing of each option is not parallelized, but each timestep of the lattice is computed concurrently to avoid long stalls.

- Despite an overall comparable throughput, the latency is much better with FIFO
- The latency improves because more workers mean that queues are overall shorter, which means fewer items are stuck behind heavy tasks. Moreover, there is more stealing, and stealing always takes the oldest elements.

Figure 5.12 shows the tail latency under FIFO and LIFO scheduling. Even though the overall throughput is comparable, the latency is significantly better with LIFO. It follows from FIFO processing each task-tree concurrently and taking advantage of terminating the shallow ones early. Significantly, different tail latencies do not suffer from this choice, as processing a small number of quick tasks before the large ones does not significantly impact the latency of the latter. This benchmark may seem specific and inconsequential to most workloads, yet this is what commonly occurs in complex systems. The distribution of tasks is not just bimodal but multimodal and thus impacts all range of tail latencies. In fact, in the limit, if some subtrees have infinite depth, other tasks may never get to run with LIFO. Practical runtime or systems schedulers typically follow FIFO to avoid

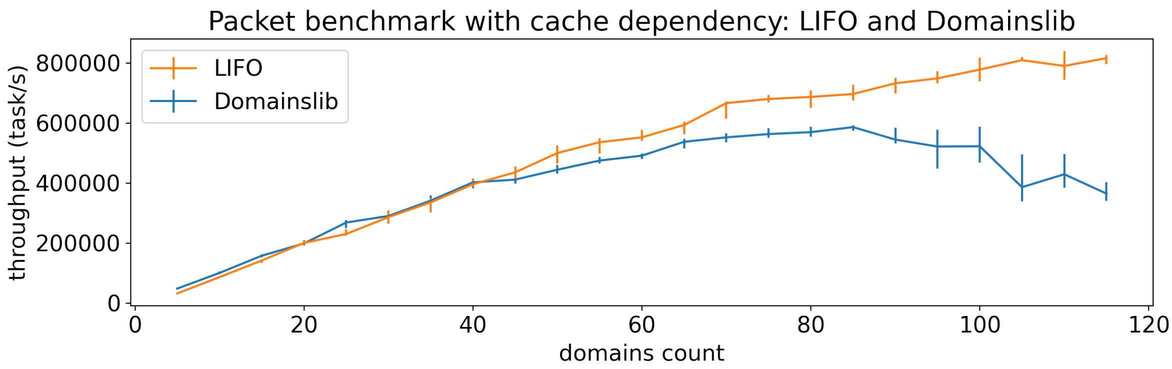

Figure 5.13: Throughput of the packet benchmark with bimodal task chains and cache dependency (§5.3.3). It used 100k packets and a single spawner. The figure shows peformance achieved by the LIFO scheduler developed in this dissertation and Domainslib [67].

starvation in heterogeneous workloads.

As more workers are added, the tail latency improves. Predominantly because the overall queue sizes are shorter (fewer items are stuck behind heavy tasks), and stealing always takes the oldest elements. However, as soon as more spawners are added, the spawning becomes more efficient and processing more local, which collapses the tail latency even with many domains.

Notably, this result contrasts with the other complex benchmark (§5.4), where the LIFO scheduler is the way to optimize latency. Furthermore, in the previous cases (e.g., §5.6), increasing the number of spawners improved both throughput and latency since it helped ease the bottleneck in work distribution. In the option pricer benchmark case, adding more spawners does improve throughput but hurts tail latency as queues grow, and processing becomes more LIFO. Again, these phenomena show the importance of customized scheduling (§1.1).

5.6 Comparison with Domainslib

Note that the LIFO in this section refers to the LIFO scheduler developed for this project. Domainslib (which is also locally-LIFO) is referred to explicitly.

The performance with Domainslib has been compared using the last packet processing benchmark (§5.3.3) as it features bimodal work distribution, cache effects and is assessed using a simple metric - throughput. Further, the benchmark was run with just one spawner to highlight the effect. It is worth pointing out that Domainslib can auto-resize, while for the LIFO scheduler, although possible, auto-resizing has not been implemented. To account for that difference, both Domainslib and LIFO were set to have a minimum size of 1024 items, which Domainslib can expand further if needed.

The results are shown in figure 5.13. The Domainslib is significantly faster at low core counts. The benefit vanishes between 20 and 40 cores, and both implementations offer similar performance. Beyond 40 cores, the Domainslib is unable to scale. It peaks at 85 cores and sees a strong throughput reduction for more cores. On the other hand, the LIFO implementation sees diminishing results but continues to improve to the last data point at 115 cores.

The detailed analysis is beyond the scope of this section, but the three key factors are described below.

- **Optimization.** The constant factors of Domainslib are more optimized. It requires only a single atomic operation per insertion or removal. Moreover, Domainslib uses several advanced techniques that are possible to apply to the LIFO implementation but are risky to use in the early development, e.g., *Obj.magic*, forcing inlining, unsafe array operations. This aligns with figure 5.13 showing a constant improvement over LIFO in the 5-15 range.
- **Scalability.** Domainslib only steals a single element at the time, while LIFO steals half of the items. LIFO only needs a single index operation per steal. Thus the actual number of atomic operations per steal of both implementations converges as more elements are stolen. However, all of Domainslib's atomic operations focus on the *head* index. LIFO, on the other hand, in proportion, uses the *head* less and less. Furthermore, stealing multiple elements creates a load-balancing network, where elements propagate through multiple hops and ease the stress on the spawner. With the steal-one policy, the load on the spawner increases linearly with the number of domains.

Chapter 6

Summary and conclusions

This dissertation focused on exploring the trade-offs between various real-world scheduling policies. The core of the work included lock-free FIFO and LIFO schedulers, which were then composed (§4.4.2) and extended in various ways (§5.3.3, §5.3.1). The evaluation involved many real-world benchmarks and standard industrial metrics: throughput and tail latency (§4.1).

The results highlight vast differences between different scheduling policies. There is no single optimal scheduling policy. Instead, it depends on the workload, available parallelism and optimized metrics. Other findings include:

- The selection of optimal policy is non-trivial. For example there is no clear-cut answer for whether FIFO, LIFO, or other is better for latency. It depends on the interaction of a number of factors, such as the shape of the task graph and locality benefits. In one case, staged architecture was shown to improve over both baseline schedulers (§5.3.3), while in another the result was opposite (§5.4).
- Scaling fine-grained workloads to high-core machines is non-trivial. An algorithm that scales well up to 20 cores can lose performance as more resources are added due to contention (§5.3.1).
- The default OCaml scheduler - Domainslib [67] - has room for improvement on high core counts (§5.6). Notably, the OCaml team has already expressed interest in upstreaming some lock-free data structures developed for this project.
- Locked overflow queue used in Go and Tokio limits performance of some workloads (§5.3.1, §5.4).

Therefore, there are clear benefits from customization and composition of schedulers for the workload and the desired metrics. Thus, the hypothesis stated in the introduction is accepted (§1.1).

Finally, OCaml multicore improves over classic runtime-integrated scheduling. Historically, built-in scheduling necessitated FIFO to avoid LIFO's bad latency characteristics under heterogeneous workloads. LIFO is, nonetheless, the optimal choice for many homogenous workloads (e.g., §5.11). The newfound capabilities of OCaml let developers tap into the benefits of using custom scheduling policies. In the case of the mixer benchmark, the homogenous mixer workload can be isolated on the LIFO scheduler, while some other diverse work would remain in the traditional FIFO scheduler.

Future work can take several different directions, e.g., further optimization of task distribution mechanisms (e.g., horizontal scaling of local structures), integration of effects and preemption, exploration of viable lock-free structures, and development of DSCheck (§2.2.5).

Bibliography

[1] 2nd gen amd epyc™ 7702 — server processor — amd. https://www.amd.com/en/products/cpu/amd-epyc-7702. (Accessed on 05/17/2022).

[2] Amd announces industry's first x86 dual-core processor. https://phys.org/news/2004-08-amd-industry-x86-dual-core-processor.html. (Accessed on 05/17/2022).

[3] How web browsers use processes and threads — by jeewantha lahiru — level up coding. https://levelup.gitconnected.com/how-web-browsers-use-processes-and-threads-9f8f8fa23371. (Accessed on 05/17/2022).

[4] Opam documentation. URL https://ocaml.janestreet.com/ocaml-core/111.28.00/doc/async_kernel/#Pipe.

[5] The Go Programming Language. URL https://go.dev/.

[6] Goroutines vs os threads. URL https://groups.google.com/g/golang-nuts/c/j51G7ieoKh4/m/wxNaKkFEfvcJ.

[7] LKML: Steve Sistare: [PATCH v3 00/10] steal tasks to improve CPU utilization. URL https://lkml.org/lkml/2018/11/9/1173.

[8] memory_order - cppreference.com. https://en.cppreference.com/w/c/atomic/memory_order. (Accessed on 05/18/2022).

[9] Transmission Control Protocol. RFC 793, Sept. 1981. URL https://www.rfc-editor.org/info/rfc793.

[10] I. Advanced Micro Devices. 2nd gen amd epyc™ 7702 — server processor — amd. https://www.amd.com/en/products/cpu/amd-epyc-7702. (Accessed on 06/08/2022).

[11] C. Agastya, D. Mechanic, and N. Kothari. Mouth-to-ear latency in popular voip clients. 2009. doi: 10.7916/D80Z794V. URL https://academiccommons.columbia.edu/doi/10.7916/D80Z794V.

[12] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for*

the Construction and Analysis of Systems, pages 41–44, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19835-9.

[13] R. Amin, F. Jackson, J. E. Gilbert, J. Martin, and T. Shaw. Assessing the impact of latency and jitter on the perceived quality of call of duty modern warfare 2. In *Human-Computer Interaction. Users and Contexts of Use*, pages 97–106. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-39265-8_11. URL `https://doi.org/10.1007/978-3-642-39265-8_11`.

[14] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999. doi: 10.1145/324133.324234. URL `https://doi.org/10.1145/324133.324234`.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '95*. ACM Press, 1995. doi: 10.1145/209936.209958. URL `https://doi.org/10.1145/209936.209958`.

[16] M. Bohr. A 30 year retrospective on dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Newsletter*, 12(1):11–13, 2007. doi: 10.1109/n-ssc.2007.4785534. URL `https://doi.org/10.1109/n-ssc.2007.4785534`.

[17] J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, Nov. 2020. doi: 10.1145/3428194. URL `https://doi.org/10.1145/3428194`.

[18] A. Castello, A. J. Pena, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Orti. A review of lightweight thread approaches for high performance computing. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2016. doi: 10.1109/cluster.2016.12. URL `https://doi.org/10.1109/cluster.2016.12`.

[19] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures - SPAA'05*. ACM Press, 2005. doi: 10.1145/1073970.1073974. URL `https://doi.org/10.1145/1073970.1073974`.

[20] chebyrash. chebyrash/promise: Promise / future library for go. `https://github.com/chebyrash/promise`, 2022. (Accessed on 06/01/2022).

[21] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic. Adaptive scheduling of parallel jobs in spark streaming. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017. doi: 10.1109/INFOCOM.2017.8057206.

[22] O. Contributors. ocaml-multicore/ocaml-multicore at 4.12+domains. `https://`

github.com/ocaml-multicore/ocaml-multicore/tree/4.12+domains. (Accessed on 06/06/2022).

[23] J. Corbet. The right way to yield [lwn.net]. https://lwn.net/Articles/31462/. (Accessed on 06/08/2022).

[24] A. Drebes, A. Pop, K. Heydemann, N. Drach, and A. Cohen. NUMA-aware scheduling and memory allocation for data-flow task-parallel applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2016. doi: 10.1145/2851141.2851193. URL https://doi.org/10.1145/2851141.2851193.

[25] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[26] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, feb 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408797. URL https://doi.org/10.1145/2408776.2408797.

[27] M. Faverge and P. Ramet. A NUMA Aware Scheduler for a Parallel Sparse Direct Solver. In *Workshop on Massively Multiprocessor and Multicore Computers*, page 5p., Rocquencourt, France, Feb. 2009. INRIA. URL https://hal.inria.fr/inria-00416502.

[28] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005. doi: 10.1109/mc.2005.160. URL https://doi.org/10.1109/mc.2005.160.

[29] HashiCorp. Nomad by hashicorp. https://www.nomadproject.io/. (Accessed on 06/01/2022).

[30] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. Chapter 1 - introduction. In M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, editors, *The Art of Multiprocessor Programming (Second Edition)*, pages 1–18. Morgan Kaufmann, Boston, second edition edition, 2021. ISBN 978-0-12-415950-1. doi: https://doi.org/10.1016/B978-0-12-415950-1.00009-4. URL https://www.sciencedirect.com/science/article/pii/B9780124159501000094.

[31] Y. Ho and D. Pepyne. Simple explanation of the no-free-lunch theorem and its implications. *Journal of Optimization Theory and Applications*, 115(3):549–570, Dec. 2002. doi: 10.1023/a:1021251113462. URL https://doi.org/10.1023/a:1021251113462.

[32] J. Hull. *Options, Futures And Other Derivatives*. Pearson College Div, hardcover

edition, 6 2005. ISBN 978-0131499089. URL https://lead.to/amazon/com/?op=bt&la=en&cu=usd&key=0131499084.

[33] C. Igel and M. Toussaint. A no-free-lunch theorem for non-uniform distributions of target functions. *Journal of Mathematical Modelling and Algorithms*, 3(4):313–322, Jan. 2005. doi: 10.1007/s10852-005-2586-y. URL https://doi.org/10.1007/s10852-005-2586-y.

[34] S. Jaffer. sadiqj/dscheck. https://github.com/sadiqj/dscheck. (Accessed on 06/01/2022).

[35] G. Kappes and S. V. Anastasiadis. A lock-free relaxed concurrent queue for fast work distribution. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2021. doi: 10.1145/3437801.3441583. URL https://doi.org/10.1145/3437801.3441583.

[36] R. K. Karmani, G. Agha, M. S. Squillante, J. Seiferas, M. Brezina, J. Hu, R. Tuminaro, P. Sanders, J. L. Träffe, R. A. Geijn, J. L. Träff, R. A. Geijn, M. B. Sander, J. L. Gustafson, R. O. Dror, C. Young, D. E. Shaw, C. Lin, J.-K. Lee, R.-G. Chang, C.-B. Kuan, G. Kollias, A. Y. Grama, Z. Li, R. C. Whaley, and R. W. Vuduc. Amdahl's law. In *Encyclopedia of Parallel Computing*, pages 53–60. Springer US, 2011. doi: 10.1007/978-0-387-09766-4_77. URL https://doi.org/10.1007/978-0-387-09766-4_77.

[37] M. Kerrisk. vdso(7) - linux manual page. https://man7.org/linux/man-pages/man7/vdso.7.html. (Accessed on 06/06/2022).

[38] H. N. Khan, D. A. Hounshell, and E. R. H. Fuchs. Science and research policy at the end of moore's law. *Nature Electronics*, 1(1):14–21, Jan. 2018. doi: 10.1038/s41928-017-0005-9. URL https://doi.org/10.1038/s41928-017-0005-9.

[39] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. In *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-39958-9_18. URL https://doi.org/10.1007/978-3-642-39958-9_18.

[40] M. Kleppmann. *Designing data-intensive applications*. O'Reilly Media, Sebastopol, CA, Mar. 2017.

[41] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *SIGPLAN Not.*, 48(8):69–80, feb 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442524. URL https://doi.org/10.1145/2517327.2442524.

[42] C. Lerche. Making the tokio scheduler 10x faster — tokio - an asynchronous rust runtime. https://tokio.rs/blog/2019-10-scheduler, . (Accessed on 06/01/2022).

[43] C. Lerche. Tokio - an asynchronous rust runtime. `https://tokio.rs/`, . (Accessed on 06/01/2022).

[44] G. Maruzzelli. *FreeSWITCH 1.8.* Packt Publishing, Birmingham, England, July 2017.

[45] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, Apr. 2022. USENIX Association. ISBN 978-1-939133-27-4. URL `https://www.usenix.org/conference/nsdi22/presentation/mcclure`.

[46] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*. ACM Press, 1968. doi: 10.1145/1476589.1476628. URL `https://doi.org/10.1145/1476589.1476628`.

[47] B. Modelski. bartoszmodelski/ebsl. `https://github.com/bartoszmodelski/ebsl`. (Accessed on 06/01/2022).

[48] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sept. 2006. doi: 10.1109/n-ssc.2006.4785860. URL `https://doi.org/10.1109/n-ssc.2006.4785860`.

[49] B. Norris and B. Demsky. CDSchecker. *ACM SIGPLAN Notices*, 48(10):131–150, Nov. 2013. doi: 10.1145/2544173.2509514. URL `https://doi.org/10.1145/2544173.2509514`.

[50] M. L. Pinedo. *Scheduling*. Springer US, 2012. doi: 10.1007/978-1-4614-2361-4. URL `https://doi.org/10.1007/978-1-4614-2361-4`.

[51] A. Postnikova, N. Koval, G. Nadiradze, and D. Alistarh. Multi-queues can be state-of-the-art priority schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Mar. 2022. doi: 10.1145/3503221.3508432. URL `https://doi.org/10.1145/3503221.3508432`.

[52] M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, Dec. 2015. doi: 10.1016/j.entcs.2015.12.003. URL `https://doi.org/10.1016/j.entcs.2015.12.003`.

[53] K. Rupp. karlrupp/microprocessor-trend-data: Data repository for my blog series on microprocessor trend data. `https://github.com/karlrupp/microprocessor-trend-data`. (Accessed on 06/08/2022).

[54] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations

on modern architectures. 2020. doi: 10.48550/ARXIV.2010.09852. URL https://arxiv.org/abs/2010.09852.

[55] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy. Retrofitting parallelism onto OCaml. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, Aug. 2020. doi: 10.1145/3408995. URL https://doi.org/10.1145/3408995.

[56] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, June 2021. doi: 10.1145/3453483.3454039. URL https://doi.org/10.1145/3453483.3454039.

[57] W. Stallings. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. 01 2005. ISBN 9788120329621.

[58] D. Stephen, W. Leo, S. KC, Y. Jeremy, and M. Anil. Effective Concurrency through Algebraic Effects, 2015.

[59] D. Sun, H. He, H. Yan, S. Gao, X. Liu, and X. Zheng. Lr-stream: Using latency and resource aware scheduling to improve latency and throughput for streaming applications. *Future Generation Computer Systems*, 114:243–258, Jan. 2021. doi: 10.1016/j.future.2020.08.003. URL https://doi.org/10.1016/j.future.2020.08.003.

[60] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA '11*. ACM Press, 2011. doi: 10.1145/1989493.1989550. URL https://doi.org/10.1145/1989493.1989550.

[61] J. Swalens, S. Marr, J. D. Koster, and T. V. Cutsem. Towards composable concurrency abstractions. *Electronic Proceedings in Theoretical Computer Science*, 155:54–60, June 2014. doi: 10.4204/eptcs.155.8. URL https://doi.org/10.4204/eptcs.155.8.

[62] G. Team. go/proc.go at 1292176bc98be4b7b9d24abec05e88b3dbd89e21 · golang/go. https://github.com/golang/go/blob/1292176bc98be4b7b9d24abec05e88b3dbd89e21/src/runtime/proc.go#L5806, . (Accessed on 06/08/2022).

[63] G. Team. go/proc.go at master · golang/go. https://github.com/golang/go/blob/master/src/runtime/proc.go#L2597, 2022. (Accessed on 06/01/2022).

[64] L. Team. tokio-rs/loom: Concurrency permutation testing tool for rust. https://github.com/tokio-rs/loom, . (Accessed on 06/08/2022).

[65] O. Team. Concurrency and parallelism design notes · ocaml-multicore/ocaml-

multicore wiki. https://github.com/ocaml-multicore/ocaml-multicore/wiki/Concurrency-and-parallelism-design-notes, . (Accessed on 06/08/2022).

[66] O. Team. ocaml-multicore/eio: Effects-based direct-style io for multicore ocaml. https://github.com/ocaml-multicore/eio, . (Accessed on 06/08/2022).

[67] O.-M. Team. ocaml-multicore/domainslib: Parallel programming over domains. https://github.com/ocaml-multicore/domainslib, . (Accessed on 06/01/2022).

[68] G. Technology. Md90-fs0 (rev. 1.1) — server motherboard - gigabyte u.k. https://www.gigabyte.com/uk/Enterprise/Server-Motherboard/MD90-FS0-rev-11. (Accessed on 06/08/2022).

[69] D. L. Tennenhouse. Layered multiplexing considered harmful. In *In First International Workshop on High Speed Networking*, 1989.

[70] Twilio. Frequently asked questions — twilio. https://www.twilio.com/docs/stun-turn/faq. (Accessed on 05/30/2022).

[71] M. Welsh, D. Culler, and E. Brewer. SEDA. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, Dec. 2001. doi: 10.1145/502059.502057. URL https://doi.org/10.1145/502059.502057.

[72] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr. 1997. doi: 10.1109/4235.585893. URL https://doi.org/10.1109/4235.585893.

[73] C. Yang and J. Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2016. doi: 10.1145/2851141.2851168. URL https://doi.org/10.1145/2851141.2851168.

[74] Q. Zhang, D. Feng, F. Wang, and Y. Xie. An interposed i/o scheduling framework for latency and throughput guarantees. *Journal of Applied Science and Engineering*, 17: 193–202, June 2014. ISSN 1560-6686. doi: 10.6180/jase.2014.17.2.10. URL https://doi.org/10.6180/jase.2014.17.2.10.

Appendix A

Rejected design choices

This section expands on §4.5 and discusses features, which did not make it into the implementation.

A.1 Backpressure

One of the primary use cases that a scheduler needs to handle is akin to the producer-consumer pattern. For example, in the case of packet processing benchmark (4.2.1), one long-lived task listens to network traffic and spawns a task per request to be picked up by other threads for execution. A common problem in systems organized in that way occurs when producer is faster than consumers, as this fills up queues. This increases mean latency and impedes responsiveness. Moreover, it effectively slows down the system due to intense memory pressure and adverse cache effects.

This can be solved using backpressure, which assumes either providing producers with an explicit signal to slow down or even blocking the producers until consumer catches up. Both approaches have proven successful in e.g. TCP flow control [9] or Async's pipe implementation [4].

Implementing backpressure in the core of the scheduler would indisputably improve some use-cases, even distributed ones. E.g. assume an operation of reading a file, sending it over network, and writing to disk. If the last step is the bottleneck, backpressure can be propagated from writing to disk to the network stack. Then with TCP, all the way to the sender, who can now allocate less resources to this operation. Nonetheless, this feature has not been implemented as coming up with a robust backpressure mechanism for a scheduler, which needs to support very diverse workloads and has local data structures, is non-trivial. It could be added as an optional feature, as is the case in the mentioned pipe implementation [4].

A.2 Cancellations

Cancellations are common in modern workloads. It is a function of more thought being given to latency but also, a convenient programming model. Program may want to do multiple requests and only process the first one to return, or at some point determine that other results are not needed. Further, cancellations are critical to recovering from an overload. Their implementation has to be particularly robust since a large wave of propagating cancellations may be the actual event to tip a system over the edge and cause meltdown. While this problem has not been considered explicitely in this work, all the core lock-free implementations behind schedulers (§5.1) allow out-of-order eviction of cancelled items, what immediately helps reduce memory footprint.

A.3 Blocking operations

As discussed (§4.5.1), developer should strive to create lightweight, fine-grained tasks for a non-preemptive scheduler to avoid latency issues. In some cases, however, it may not possible. A long system-call may not have an alternative, non-blocking API. While blocking is unavoidable, scheduler may offer still offer some support.

- One approach is to simply let user manually specify long-running operations and have them scheduled on a new system thread (§3.2). Such approach suffers from the same disadvantage as Async/Lwt, as in developer will occasionally fail to annotate a function.
- Alternatively, scheduler may feature a monitoring thread, which automatically adds new threads to the pool whenever there is a risk of starvation.

This has not been explicitely handled in the current implementation but both approaches are feasible. There is a question of how important is such a measure, given that the proposed IO library (eio [66]) tries to use the fast, non-blocking interfaces like io_uring.

Appendix B

Core data structures

This section describes the underlying lock-free structures.

B.1 MPMC queue on infinite array

As the first step, it is crucial to understand the main idea behind any multi-producer or multi-consumer FAD-based queue. A particularly intuitive exposition is offered by [73] (which takes a wait-free direction afterwards). The key simplification is to assume that said queue is built on top of an array of infinite length.

Functions operating on such an array are presented in listing B.1 and B.2. The enqueuer simply increases the *tail* index by one using an atomic operation, receives the index of its slot in return and stores the enqueued element there. Since the queue is infinite, every slot is only used once. Likewise, the dequeuer begins by increasing the *head* index and receives the index of its slot in return. However, since there is no synchronisation with the tail, the actual slot can still be empty. Either because head overshot the tail or enqueuer is simply slow to finish operation. Thus, dequeur needs to make sure that the slot actually contains an element and once that's established, it returns the element.

With such a setup both enqueue and dequeue need do two atomic operations (FAD+GET/SET) but the contention is resolved at the *head*, *tail* indices with FAD. Thus, there is never any need to retry in the hotspots. Once past the FAD, each dequeuer is essentially matched up with an enqueuer and a slot to pass the information through. Accesses to the slot need

```
let enqueue {array; tail; _} element =
  let index = Atomic.fetch_and_add tail 1 in
  let cell = Array.get array index in
  Atomic.set cell (Some element)
;;
```

Figure B.1: Basic enqueue function.

```
let dequeue {array; head; _} element =
  let index = Atomic.fetch_and_add head 1 in
  let cell = Array.get array index in
  while Option.is_none (Atomic.get cell) do () done;
  Atomic.get cell
;;
```

Figure B.2: Basic deque function.

to be atomic as enqueuer and dequeuer may work in parallel. This extra atomic operation is avoidable in some cases and convenient in others (e.g. implementation of cancellations A.2 or lightweight yielding 4.5.3). Overall, it is a constant factor and does not impact scalability of the structure.

B.2 MPMC queue blocking

While the infinite array is a powerful idea and some designs actually simulate that using a linked list of arrays (e.g. [60]), use of circular buffer is far more common. Luckily, the design described in the previous section almost works on a circular array. The major difference is that now enqueuer wraps around the array and may encounter a full slot, thus enqueue function requires some mechanism to not overwrite another item, e.g. a busy wait akin to that in figure B.2. Secondly, both enqueuer and dequeuer must use CAS to edit the item slot as the queue can overlap multiple times.

Such an implementation constitutes a fully-functional multi-producer multi-consumer unbounded queue as described in [35]. Importantly, such an implementation remains somewhat inconvenient for a modern system. Busy waiting is generally bad for performance. Especially since this busy waiting is not the typical busy waiting in lock-free algorithm, which is expected to take a few tens of cycles. Further, such a structure is no longer strictly FIFO as two overlapping enqueuers or dequeuers are essentially racing on the same slot. Both issues are fixed in the following sections.

B.3 MPMC queue non-blocking dequeue

Using FAD eliminates the contention in dequeue() but it trades-off consistency. Now, dequeue() may increment the *head* only to discover that is has over overshot the *tail*. In fact, multiple threads can overshoot at once causing the *head* to be multiple slots higher than *tail*, where the maximum discrepancy is bounded by number of independent processing units. There is a number of possible ways to recover from an overshoot.

- Use CAS to rollback the increment. The CAS operation has to precisely revert the increment done by FAD. The CAS-rollback can fail if there is another dequeuer than incremented the *head* even further. There are two possible cases. Firstly, it is going to try to rollback too and said CAS should be retried. Alternatively, in the meantime enqueuers have appeared and the other thread actually found an item in the higher slot. In such a case the latter thread is not going to rollback but the current slot has a paired enqueuer and should have item shortly.

- Analogically, CAS could be used to increment the *tail* to cover the overshot. This has the benefit that overshooters free themselves in FIFO order but adds load onto the *tail* index and negatively affects enqueuers in a busy loop.
- Mark slot as burned and return immediately. Then once enqueuer gets assigned that slot, it discovers that dequeuer is no longer active, recycles it by removing the mark and retries the insertion from scratch. Again, this puts extra stress on the enqueuer and requires additional logic to prevent too much of the queue to be marked (as it inflates queue true size).

The implementations in this project rely on the first solution. The third one is useful in the wait-free algorithms as it helps bound maximal number of retries of other methods.

B.4 MPMC queue non-blocking enqueue

Similar rollback methods apply as to the dequeue. Moreover, items can be queued up as a linked list in a single slot (what may also be used to bring back strict FIFO order) or the enqueuer may simply resize the queue. There is perhaps an additional consideration from the perspective of work-distribution, if the queue is already overloaded, one should not cause additional rollback synchronisation and cause further slowdown (in e.g. passing the tasks to overflow queue).

For the purpose of this dissertation there was no need to use any of these methods. MPMC is only used as the global overflow structure, which generally means that it is not commonly full and at that point putting halt on the spawner is likely better than letting it allocate endless memory. The local work stealing queues, on the other hand, are single-producer, which means that the enqueuer is able to use additional assumptions to avoid the need to rollback.

B.5 Work stealing queue

In the case of work stealing, there is no need to employ a full MPMC queue. Local queues only have a single enqueuer and that eliminates the possibility of races on the *tail* index. This has two important benefits.

- Any accesses to *tail* index have to be atomic as other functions read *tail* but updates may occur in multiple steps, i.e. atomic get, modification, atomic set. Such updates are cheaper to execute than fully-fledged atomic updates using FAD or CAS.

- The single enqueuer is able to ensure that it is not going to overlap the queue. It can simply check whether there is space in the queue and act accordingly as no other enqueuer can act in between. A natural single-core approach would be to simply check whether *head* index is sufficiently far. But this does not guarantee that the dequeuer has already removed item from the slot. Thus, an easier and more approach is to simply check whether the slot in *tail+1* is empty. If it is, enqueuer can safely place an item and increase the *tail*.

That gives rise to a fast local enqueue. It is an ideal choice for a work stealing system, in which threads under heavy load focus entirely on their local work. Likewise, to fully optimize the local queue, it would be nice to have a fast local dequeue. Again, such a local dequeue can utilize the knowledge that no new elements are inserted during its execution. Therefore, it can begin by executing a FAD operation on the *head* index and if it has overshot the *tail*, it necessarily has to roll back.

Possible rollback methods have been described in §B.3. None of them is perfect. In particular, rolling back any of the indices may be stuck behind other overshot dequeuers. Potentially paralylizing the local dequeuer until another thread gets scheduled. Thus, the actual SPMC Queue implementation has two dequeue methods:

- **Local deque.** It takes a single element from the local queue. It is the only method that is allowed to speculate (and possibly overshoot) using FAD. Therefore, whenever local dequeue overshoots, it is guaranteed that no other thread can overshoot further. With such knowledge the rollback is trivial. Local deque simply sets the *head* to what is used to be, and terminates. Local deque only needs to synchronize with steal function.

- **Steal.** It attempts to transfer half of the elements of the victim's queue to its local structure. To simplify the local dequeue logic, it is not allowed to speculate on the *head* index. Instead steal uses a CAS operation, which guarantees no overshooting as *tail* cannot decrease. CAS is naturally slower and more contentious but, importantly, there is only one CAS operation required to take many elements. Therefore, the extra cost of index CAS is in effect very small on per-item basis if queue had a considerable number of items in it. Notably, steal can be called from any thread and needs to synchronize with both local enqueue and local dequeue.

Such a design gives rise to a practical implementation of SPMC structure. It is scalable and follows the philosophy of work stealing by prioritising local operations.

B.6 Work stealing stack

Assume that a stack with synchronized local functions exists. A natural question to ask is whether the steal should also follow LIFO ordering or just take the elements from the top of the stack (FIFO). The latter has two considerable benefits over LIFO. Firstly, FIFO steal means that stealers contend for the bottom of the stack, while local thread works only the top. Such a setup is very convenient since local and global methods work on different indices, what reduces complexity and contention. Secondly, the alleged benefit of LIFO is locality. However, a different thread is not going to benefit from that as much as the owner. Thus, instead, it can take the oldest elements in the stack (FIFO) and significantly improve tail latencies.

Fast local push and FIFO steal are already implemented in the previous section (§B.5). The only missing bit is local (FIFO) pop. One way to implement it is akin to the FIFO push, i.e. firstly act on the array and only update the *top* pointer afterwards. With such a procedure, local pop simply tries to remove item from the array. It either succeeds (and updates *top*) or not, and terminates. Stealer, on the other hand, may have its item taken by pop operation, and needs to be able to roll back.

This design is similar to the classic work stealing deque [19]. However, this implementation:

- Steals half of the queue rather than a single element. Stolen elements are transferred directly into the thief's queue rather than returned.
- Local pop does never needs to rollback an index. Instead, it executes CAS on a less contended item slot.
- Has the extra flexibility coming from atomic slots. That helps with implementation of e.g. cancellations (§A.2) and soft yields (§4.5.3).

B.7 Resizing

The work stealing queue has been extended to allow resizing. It lets the owner of the queue increase the size of the local structure rather than push overflowing elements into a separate queue. This, in theory, may translate to better work distribution. Such a functionality may seem daunting to implement in the lock-free world but, curiously, the implementation of FIFO work stealing lets the owner lock out all other users.

The mechanism relies on the fact that non-owners do not speculate when stealing. Therefore, owner can simply overshoot the *head* index to ensure that no new stealers appear. Once the *head* is higher than *tail*, owner needs to wait for all in-flight stealers to terminate. It can detect it by checking whether all *claimed* elements have been removed from the array. Once that's done, the elements are transferred to a new array, the array and mask are updated in the record, and then *head*, *tail* indices are carefully corrected.

B.8 Sharded queue

Performance of staged architecture (§4.4.2) and overflow queue depends on the scalability of the underlying queue. Lock-free MPMC queue (§B.2) allows enqueuers and dequeuers to work faster and more independently than with a locked structure. However, the throughput of such a queue hardly improves beyond more than a few workers.

A common technique used to extend the scalability in such circumstances is horizontal scaling. Instead of a single queue with two points of contention (indices), the structure can encapsulate multiple queues and load balance across them. Such a construction is called multi queue and can alone constitute a core structure of a scheduler [51]. This dissertation uses a simple multiqueue wherein enqueuers insert items into randomly selected MPMC queue, and dequeurs start with random queue and iterate until a non-empty queue is found (or all queues have been checked).

B.9 False sharing

A common optimization pattern in low-level programming involves separating contended variables into separate cache lines to ensure they can be owned independently. This was taken into consideration only in the places where it was natural (e.g. size of runqueue). OCaml does not allow to explicitely specify padding or memory layout of a record type. It is possible to force some particular layout by adding unusued variables in crtitical places but that would be quite fragile as compiler updates or changes to the record itself may impact the layour in unpredictable ways. Thus, implementations were not optimized for such subtle behavior.

B.10 Validation

The core lock-free queue and stack have been extensively load tested and verified using DSCheck (§2.2.5). DSCheck has been modified to put a hard limit on exploration of infinite loops in algorithms requiring busy-waiting. The modified version is vendored in the main repository [47].

B.11 Limitations

Since this project has a limited timescale, the schedulers were optimized for scalability but their implementations do not eliminate all constant factor overhead. In part since these overheads are not critical to the hypothesis (§1.1) but also because it requires a far more error-prone code in OCaml.