UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Research project report title page

Candidate 1234N

"A LaTeX template for a Master's project report"

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

Total page count: 54

Main chapters (excluding front-matter, references and appendix): 41 pages (pp 8–48)

Main chapters word count: 467

Methodology used to generate that word count:

[For example:

```
$ make wordcount
gs -q -dSAFER -sDEVICE=txtwrite -o - \
   -dFirstPage=6 -dLastPage=11 report-submission.pdf | \
egrep '[A-Za-z]{3}' | wc -w
467
```

]

Abstract

Write a summary of the whole thing. Make sure it fits on one page.

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

20 years ago computer processors looked very different than today. The first stock, non-embedded multicore processor has just been released [5] but the customers would have to wait until 2004 to see the first multicore x86 CPU - dual core AMD Opteron [2]. Up to four of them could have been stacked inside HP ProLiant DL585 server. At the time, a revolutionary technology, which allowed continuation of the exponential increase of computational power in chips [23] 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 equipped 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 in the daily life of programmers and in research, in algorithms oftentimes claimed to be proven [36]. This inability to take advantage of parallelism is suggested to have a real impact on the productivity of IT-using sectors of economy [30]. There is a number of research spaces trying to make parallel programming easier. One particularly important for modern operating systems and runtimes is scheduling.

A common definition of scheduling explains it in terms of allocating resources to tasks. Justifiably. 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, computer could get by with far less advanced schedulers as hardware offered no parallelism. Similarly, workloads used to be less concurrent. With less networking, worse interfaces, fewer programs running on a single computer, more acute cost of context switching, interactivity and fairness were a far smaller concern than they are today.

This is no longer the case with processors scaling mostly horizontally and evermore 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, there are significant performance advantages to be gained from customizing and composing schedulers.

The first part of the statement is akin to construction of No Free Lunch theorem [27] for schedulers. While there is a growing body of NFL theorems for optimization problems (e.g. [48], [28]), such fundamental results typically 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 point of view of hardware parallelism (1.1.1), software concurrency (1.1.2) and the opportunity presented by OCaml Multicore (1.2.2).

1.1.1 Parallelism

Moore's law is an observation stating that number of transistors in integrated circuits tends to double every two years [35]. Around the turn of the millennium, increasing single-core performance at exponential rate was no longer possible due to physical limits. Thus, manufacturers turned to stacking multiple processing units within a single chip [25]. As shown by `https://github.com/karlrupp/microprocessor-trend-data`, the exponential growth of transistors continued, together with exponential increase in cores. While some researches conjecture this trend to slow down [22], [23], we still see rapid, if not exponential, growth in this area.

create figur

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

- Designs need to evolve as synchronization primitives like locks or atomics do not scale endlessly and e.g. a naive work stealing scheduler may have been good enough on 16-thread Intel Xeon in 2012 but fail to utilize all 128 threads of a contemporary AMD ThreadRipper [18].

- Modern high-core architectures feature non-uniform memory. Thus, memory latency patterns vary with the topology and scheduling decisions may benefit from taking memory hierarchy into account (e.g. [21], [24]). Moreover, the non-uniformity appears also in consumer products such as Apple M1 or Intel Core i7-1280P. These feature two sets of cores: one optimized for performance and another one for efficiency.

1.1.2 Concurrency

Workloads become more concurrent and there is a number of reasons behind that. High concurrency is a necessary condition to take advantage of the available parallelism. Compute-heavy application have strong incentives to try to distribute work across multiple cores in the world of stalled single-clock speeds.

Secondly, the complexity of underlying workload increases as computers become more ubiquitous. There is more useful programs available and the act of composing their executions (to run together) requires concurrency. The programs alone become more complicated. Data-intensive applications [32] require more concurrency as networked portions of the code benefit from scheduling guarantees.

Finally, concurrency is a convenient tool for expressing certain workloads. Legacy Windows GUI applications are organized around an event loop, which processes inputs from the user. Programmer has to carefully judge the handler logic to never starve the loop and cause the UI to freeze. It turns out very difficult in practise; 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. A modern Go application can simply spawn a new goroutine for every handler and rest assured that runtime prevents starvation of the event loop.

1.2 Tools

This sections 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. Used system features two AMD EPYC 7702 processors, which gives a total of 128 physical cores and 256 threads. It runs Ubuntu 22.04. For more details, see .

o methods

1.2.2 OCaml Multicore

Scheduler as a library, scheduler composable

OCaml is a unique functional programming language, which aims 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 the development in a number of ways, e.g.: there is no way to efficiently switch between tasks, tracing becomes harder as there is no real notion of separate threads and stacks.
- **Blocking.** Combine single-core with lack of preemption and any single CPU-intensive or blocking call is prone to cause chaos by starving other threads: cause network timeouts, delay timers, etc. On the surface, there is no qualitative difference with multicore as any n non-preemptive threads can by blocked by n misbehaving task. But in practise multicore is far safer. Firstly, 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, which spawns new OS threads whenever needed [7].

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

Finally, Multicore OCaml is a particularly interesting object of study as it includes a number of novel solutions. Common programming languages either include threading support, which is tightly coupled with the language itself, or offer no support and, 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 allows composing schedulers. A much-needed mechanism for executing diverse workloads with portions having different optimization criteria [46].

1.3 Findings

fill in

Chapter 2

Background

This section begins by describing the general problem of scheduling, key concepts and two modern real-world runtime schedulers (2.1). Afterwards, it describes the tools of choice: important properties of OCaml programming environment (2.2) and details of fine-grained multicore programming (2.3).

2.1 Scheduling

Scheduling attempts to assign resources to tasks in a way that optimizes some metrics [37]. The problem itself is not limited to computer science and arises in a number of 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 existence of these pauses by saving and restoring 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. This decision-making perspective is illustrated by figure 2.1 and 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, which allows to detach the number of synchronous executions from the number of processing units. All kinds of program grew to rely on this abstraction heavily. For example, Google Chrome spawns a single process per tab [3], and depending on the logic, each such process spawns additional threads. This aggressive acquisition of resources leads to smooth operation of the web browser but puts significant pressure on the operating system, which has to ensure that in the sea of

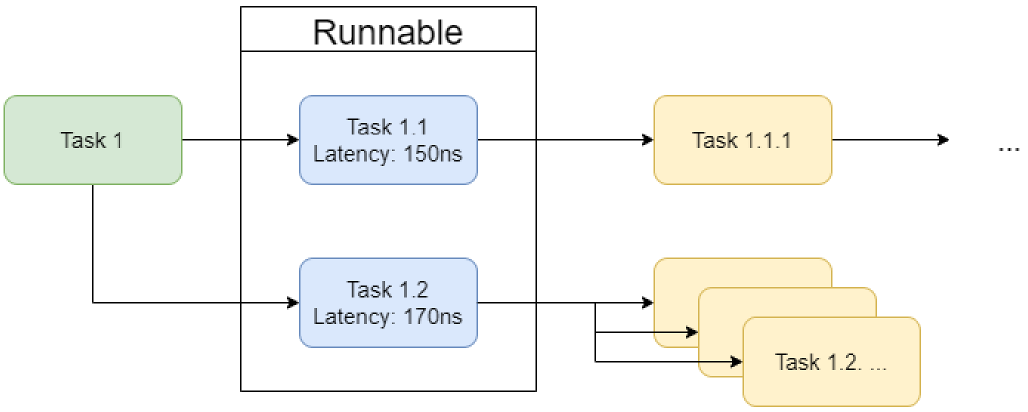

Figure 2.1: A simplified graph of fine-grained tasks, which scheduler attempts to traverse in order to terminate. Green box represents executed task, blue boxes are currently enqueued tasks. The orange boxes are tasks that are going to be scheduled once their parent execute. They are not visible to the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

threads everything gets a fair share of the CPU time and their individual objectives are achieved, e.g. a heavy web browser and light but latency-sensitive audio driver.

2.1.2 Lightweight Threads

System threading provides the fundamental mechanisms to execute concurrent workloads across parallel hardware. But programmers need to be wary of designed computations as the cost of interacting with kernel is non-trivial. It may vary from mode switching to full syscall overhead. System call logic requires a number of things to happen (such as a, b, c) and that easily thwarts the cost of a small task. Thus, the performance of highly concurrent workloads suffers significantly.

This is inconvenient for the developers. Spawning little concurrent tasks is the natural way to express many common server-side workloads, e.g. processing HTTP requests. Therefore, language designers started to construct lightweight threads, what stands for threading implemented entirely in the user space. They aim to be very cheap to use and they do deliver on that promise. Context switch in Go requires saving and restoring just 3 registers [8]. In OCaml only 2 [42]. Conversely, a system scheduler needs to save and restore *all* registers and also perform mode switching [8]. Therefore, lightweight threads are naturally better suited to support fine-grained concurrency. The next section touches on a disadvantage of not having access to privileged mode 2.1.3.

could say that there i a philosoph ical differ- ence as OS learns abou long-lived threads. ta are small a done quick

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, in which each program is assigned its quantum of CPU time. Once the quantum is used up, scheduler forcibly ensures other programs get to run. Preemption is a necessary condition to keep the system stable in the world with imperfect and malicious programs. In practise, preemption has more profound consequences. Individual programs no longer have to worry about their composition with others. They may cooperate but ultimately, all decision-making is done by the privileged scheduler, which is able to make more holistic decisions.

Nevertheless, true preemption is not possible as a pure software solution. Operating systems rely on the hardware support for privileged mode and interrupts. After all, if preemption was not restricted, a single erroneous or malicious application could completely destabilize whole system. Alternatively, running runtime of business logic within the kernel is not acceptable as such application threatens entire infrastructure if compromised. This makes true preemption inaccessible to lightweight threads for security reasons. Moreover, even if there was a safe mechanism to allow true preemption without elevating preemptor into privileged mode, such mechanism might defeat its purpose. Lightweight threads aim to be very cheap and that conflicts with using full OS-level context-switching mechanisms.

Different programming languages and libraries deal differently with this limitation. Particular approaches are discussed in further sections (3.1, 3.2).

2.1.4 Work stealing

Work-stealing is an extremely widespread concept applicable to all kinds of multicore scheduling. It has been 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) [9], Cilk [15], Go [7].

Work-stealing splits the global work queue (in the FIFO case) into a number of local, per-CPU runqueues. Each processor strongly prioritizes its own runqueue; namely, it will enqueue and dequeue elements from its own structure as long as it is possible. This alone is similar to a number of unrelated processors, actual distribution of work happens when the impossibility criterion is met. If enqueue is impossible (due to lack of space), owner can either resize the runqueue or transfer some of the work into a global overflow queue. If dequeue is impossible (due to lack of items), processor chooses a different processor at random and steals work from the victim, thus balancing the load. There is a number of benefits flowing from such organization.

- **Less synchronization and faster hot path**. Since local runqueues are single-producer, the enqueue operation needs little synchronization. Likewise, local dequeue operation only need to synchronize with stealers making it fast as well.
- **Low contention**. Whenever load on the system is high, processors converge to using their own queues, thus eliminating any need to compete with others for work.

- **Locality**. Children tasks are likely to execute on the same processor as their parents, what maximizes the locality and improves overall performance.

Chase-Lev deque (double-ended queue) [19] is one of the notable, classic work stealing implementations. Its design is similar to the example describe above but while steal remains FIFO, local insertions and removal follow LIFO order. It aimed to minimize required synchronisation, since local thread operates at one end of the structure and steal at the other but, incidentally, also improves locality of the workload 4.2.

mention de

2.1.5 Summary

Built-in scheduler - better introspection but cannot be plugged in.

Library-solution without effects, sure but it quickly ends up cumbersome to the point where we have ppx rewrites to actually fill in the monadic handlers around every statement. Further, combination of monads becomes cumbersome.

Existing schedulers, how they are built in and what not.

Composability and things

2.2 OCaml

As touched on in 1.2.2, OCaml introduces a new flavor of multiprocessing. It brings a number of benefits, but the most important one from the point of view of this dissertation is decoupling of scheduling and runtime [43]. In simple words, a developer of an external OCaml library now has the ability to write a powerful, built-in-like runtime scheduler without requiring any changes from the standard library, the runtime system or OCaml compiler. The rest of this section describes mechanisms enabling this decoupling and other notable features of Multicore OCaml.

2.2.1 Threading

Starting with version 5.0, OCaml will offer access to OS threading through the domain interface. System threads are widely considered heavyweight due to high memory footprint and the overhead of systemcalls . Domains are similar, in that a single domain consist of an OS thread and additional OCaml structures such as local minor heap [41]. Generally, domains offer course-grained parallelism and are optimized for the number of domains to not exceed number of processing units (`https://github.com/ocaml-multicore/ocaml-multicore/wiki/Concurrency-and-parallelism-design-notes`).

citation

To use system threads efficiently, we also need concurrency (see). Historically, OCaml did not provide any explicit support for writing concurrent programs. Thus, developers using

concurrenc
section

Lwt/Async are forced to divide programs into small functions, which are then scheduled dynamically and executed on the same stack.

Multicore OCaml features fibers - light, heap-allocated, resizable stacks. Their count is mostly limited by memory and, in general, may exceed number of logical cores by orders of magnitude without any issues. Importantly, context-switching between fibers is extremely cheap. It requires saving stack and exception pointers of the fiber and loading ones of the target frame [42]. Therefore, they are a perfect foundation for efficient, fine-grained concurrency.

2.2.2 Effects

Effects represent a novel way of modelling not just concurrency but all kinds of computational effects. The concept is a product of research on theory of computation [39] and, even though the formal underpinnings are far beyond the scope of this report, it is introduced the easiest by analogy.

An imperative programmer may consider it a generalisation of exceptions [17]. Using exception language, 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 on [42]. This is especially important for concurrency, as with Async and Lwt real-world applications end up having almost all code written in the monadic form to prevent liveness issues.

In the case of threading, effects provide a composable mean of stopping and resuming execution. Once effects handlers are set up higher up the stack, any function may perform a yield effect, which immediately returns the control back to the handler. Up until now, the logic is implementable with exceptions. However, the handler also receives a *delimited continuation*, which resumes execution of the yielding function. In the case of a scheduler, handler may put received continuation into a queue, take the oldest element and resume it instead. This logic is implemented efficiently using fibers described in the previous section (2.2.1). Whenever effects handler is set up, it instantiates a new fiber for contained logic. If said logic performs an effect, said fiber is unchained from the top of the stack and a new one is created to execute the handler. Naturally, this harmony is by design, and fibers and effects have been strongly optimized to support high-performance concurrency.

It is worth pointing out, that in the current shape, effects do not mix with preemption. Generally, true preemption is limited to OS-level scheduling and there is no way around this restriction, as running ordinary applications with higher privileges is not acceptable from security standpoint. Runtime systems typically simulate preemption by having compiler automatically insert optional yields into strategic places (checkpoints) in the code. Assuming that compiler provides us with a mean of registering a checkpoint-handler, there is no obvious solution for auto-yielding, as trying to yield within such a handler means that suddenly any line in the code may emit Yield effect - even a null function. Further, user may compose multiple schedulers and pollute all function types in the program, what defeats the point of effects system.

2.2.3 Memory model

Other languages depend on global data race freedom. That does not facilitate reasoning about programs if a race in one library may cause undefined behavior in a different one.

OCaml Multicore features a local data race freedom property, which guarantees that races are bound in space and time. fill in

2.2.4 Garbage Collection

Could mention a bit, as some charts actually explain the performance by promotions. fill in

2.3 Multiprocessing

Multiprocessing is a crucial technique to take advantage of the modern processors. There is a number of multicore programming paradigms, each providing different trade-off between safety and performance. While it is rarely the case in software engineering practise, schedulers are critical enough to fully prioritise the latter criterion, and choose lock-free programming.

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

2.3.1 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 almost boils down to using relevant threading API, e.g. computation of five independent hashes. In fact, system might automatically parallelize a computation specified by *List.map* [4].

However, on the other hand of the spectrum lie programs, for which parallelization yields marginal or no benefits. It is enough to slightly tweak the previous example to show that. Assume computation of a chain of 5 hashes, wherein each hash depends on the output of the previous one. In such a case, the hashes themselves need to happen in a sequential manner. There might still be opportunity for parallelization but it depends strictly on implementation of the hash operation. Moreover, 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.

The speedup coming from parallelization is described by Amdahl's law [26].

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

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 coming from using multiple processors and hints why adding more cores yields diminishing returns, as observed in the latter sections. Namely, doubling the number of processing units halves the time spent in parallelizable portion of the workload. Thus, gains on parallelizable portion rapidly become smaller than any non-trivial sequential part of the program.

2.3.2 Lock-free algorithms

Lock-freedom characterizes algorithms, which guarantee that at least one of many threads is going to terminate in a finite number of steps [26]. This is not the case with locks; if thread holding a lock is not scheduled then no one is allowed to make progress. Lock-free algorithms are built with 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, 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 in the form of better control over synchronisation, which may translate into better performance. Naturally, performance improvement is not guaranteed; contentious lock-free algorithm may scale much worse than an alternative implementation based on high-quality locking.

2.3.3 Atomic operations

Atomic operations are hardware-supported instructions, which guarantee synchronized, indivisible execution. They are mostly non-blocking and tend to 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 3 arguments, one variable (memory address and two values: old, new. If the variable equals the old value, then it gets updated to the new value in one step. Success or failure are reported 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 prior value is reported. FAD always succeeds.

Both operations have similar latency when applied to the same use case [40]. 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 certain kinds of memory reordering [10]. OCaml atomics only come with sequential consistency, which is the strongest restriction. Sequential consistency forbids any reordering across an atomic operation. Therefore, memory order relaxations are not further considered in this report.

2.3.4 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 in parallel, be it in a distributed system or on architectural level. High level concurrency primitives such as locks help to achieve this property trivially; if relevant state is protected by a single lock then then any functions linearize as they are forced into sequential order by the very lock. However, such coarse-grained primitives may leave performance on the table.

When performance is a priority, a common approach is to try to distribute the state to allow many agents to act in parallel. Here, linearizability is fairly easy to involuntary give up, and becomes a critical property to monitor when verifying correctness. For example, queue allows enqueuer and dequeuer to operate independently on different ends. Thus, a performance can be nearly doubled by using atomic operations to modify *head* and *tail* indices independently. However, suddenly, a simple size function *tail* - *head* may produce gibberish output as it no longer linearizes. As show in the figure , independent reads of *head* and *tail* may allow *tail* - *head* values not indicative of actual queue size.

figure

2.3.5 LF queue

maybe fill i
or leave for
impl. detai

2.3.6 MultiQ

maybe fill i

`https://dl.acm.org/doi/pdf/10.1145/3503221.3508432`

2.3.7 DSCheck

Parallelization is an error-prone process. Humans are simply not good enough at the kind of precise reasoning required to deal with super-exponentially increasing number of interleavings as more functionalities are added. The tools available until recently were also of little help.

Assumes determinism. Gives an opportunity to explore extremely rare paths.

n

Note stronger memory model 2.2.3.

cribe dis-
utions?

2.4 Benchmarks

This section describes the three main benchmarks used to evaluate different designs of a scheduler. They aim to simulate some true 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. This significantly reduces noise in the benchmarks and improves replicability.

Other benchmarks are introduced in the latter sections of this report, but they tend to be more artificial and targeted at exposing particular differences between scheduling strategies.

2.4.1 Packet Processing

TCP constitutes is a 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, algorithms for processing TCP packets became very sophisticated. To minimize executed instructions and memory accesses, they tend to only parse the parts of the packet that are necessary to proceed. This is obviously crucial for performance but also security, as e.g. flooding packets should be recognized and dropped with as little overhead as possible.

Therefore, the first benchmark simulates workload similar to a webserver. Firstly, it accepts a packet and spawns a lightweight thread for its processing, which in turn scans for newlines - step necessary to do more selective parsing afterwards.

ld add a
e more
kground
why this
nportant,
vious work
ust details

2.4.2 Audio Mixer

Telecommunication is a relatively low latency endeavour. Research in human-computer interaction generally recommends keeping latency of (any) response below 100ms for it to feel instantaneous ([34], [13]). Beyond that, users begin to notice the delay and the experience deteriorates. [12] 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 but there is a lot that can go wrong (and goes frequently, as we have experienced during the pandemic).

- **Network latency**. In practise vast majority of the RTT budget is consumed by the transport. Packet may have to cross multiple carrier networks or simply a large geographical distance. In practise, the less latency is imposed by the server the better average user experience.

- **Throughput**. Typical VoIP call-leg sends 20ms frames, each containing a few hundred bytes of audio. In my professional experience, a typical FreeSWITCH [33] installation was expected to handle up to 1000 concurrent calls without quality loss. 1000 concurrent calls with 2 legs each and bidirectional media means processing 100k packets a second. Each with a real deadline much smaller than the mentioned 150ms.
- **Audio complexity**. Moreover, standard VoIP signalling protocol SIP allows every participant of a conference to negotiate different codecs. Thus, the server has to decompress all incoming audio, mix it together, compress, and only then send out. In the case of mentioned 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 a number of conferences and participants, and FreeSWITCH-like audio-processing, audio-mixing threads. This benchmark aims to simulate an enterprise application with extremely complex scheduling patterns and a perhaps not perfectly optimized code, where individual threads frequently need to yield to wait for others.

2.4.3 Option Pricing

Interesting task-spawning pattern here.

Option pricing -¿ DP Smith waterman is almost the same as option pricing

Chapter 3

Related work

3.1 Go

Go is an imperative programming language, which aims to be the language for building distributed systems. The first sketches of Go were created by senior Google engineer, heavily drawing on company's extensive experience with building complex backend applications. Go features a deeply integrated support for lightweight threads, called goroutines, whose design was central to the language since its very beginning.

Go standard library adds a significant amount of machinery on top of the POSIX thread interface. Goroutine is quite 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. In contrast with typical OS threads, Go works hard to let developer schedule new routines as freely as possible, allowing hundreds of thousands of them to be executed concurrently thanks to e.g. marginal memory footprint - 2KiB per routine at initialization, while OS threads costs 1MiB (although the memory is virtual).

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 (synonymous to a single processing unit) rather than worker (OS thread). This decoupling improves remarkably over a naive architecture with runqueue per OS thread. Firstly, task may not be able to make progress if in a system call or a synchronization primitive. In such a case worker constitutes the minimal state that processor has to park if blocked and unpark if runnable. Secondly, attaching runqueue to a processor maintains the local invariant (even though runqueue changes hands) and keeps the overall number of queues much lower. Thanks to this rotation there is no tasks lingering in inactive runqueues and necessitating more stealing. On the other hand, if worker runs out of work and decides to steal there is less fragmentation and lower chance of selecting other empty queues. Finally, with such a design autoscalability emerges very naturally as it boils down to minimizing the number of runnable workers within available resources.

An obvious disadvantage to this setup is that this threading model is forced upon user, who at best gets a few knobs to tune. Even though the language is multi-purpose at heart, incurring these threading overheads might not be justifiable in embedded devices or in high-performance computing. Thus, Go is unlikely to conquer other niches.

Add more

- LIFO slot and mention of LIFO as future work in the design doc.
- Overflow strategy (non-resizable runqueue).
- Fancy features for liveness and deadlock detection.
- Randomized select to combat hyrum's law

Nice graph with proces sors, worke and tasks.

3.2 Rust

Rust is another modern imperative programming language that, nonetheless, 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 handed to the garbage collector (safe, non-trivial overheads). Rust strives to achieve both: safety and performance on par with C++. It applies a number of novel ideas, e.g. ownership system, but 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.

Therefore, threading in Rust's standard library largely maps to OS threads and leaves building out more advanced management to the user. One such attempt is Tokio. An external library, which provides multithreaded runtime and an asynchronous version of standard library.

Just like Go runtime, Tokio also relies on work stealing to share work. However, it does not abstract out OS threads and relies on a much simpler scheme with a single worker per OS thread and number of OS threads corresponding to number of processing units. As the total number of threads is limited and 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.

Touch on:

- Internals (multiple SPMCs).
- Development of Loom model-checker!
- Notably, with no runtime support the entire API is entirely function-based, so there's no natural way to do a yield.

Chapter 4

Design

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

4.1 Objectives

During the early days of computing, throughput of a system was significantly more important than the interactivity. After all, 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 human operator. Over time, however, computers became ubiquitous and networked. People and businesses started to depend on automatic processes terminating timely and reliably. In fact, major software companies such as Amazon, Google have measured the direct impact of latency on their revenue [32].

Therefore, in line with the literature (e.g. [20], [49], [44]), 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 over some workload. Naturally, 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 99.9th percentile of latency and deemed higher ones not cost-effective [32]. But a test-suite of a distributed system may consider both 99.9th percentile and median to have a better picture of the system, while a low-latency market maker may focus primarily on top 20% 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 explicitely 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 is going to have a higher impact on mean or (sufficiently small) nth percentile latencies. However, as OCaml does not feature any preemption mechanisms (§[**?**]) and the CPU time is effectively specified by the developer, the schedulers developed in this project treat all tasks as drawn from the same distribution. As discussed further in (§4.3.2), staged architecture does allow developer to more explicitely choose between fairness, latency and throughput.

4.2 Ordering

This section describes the core ordering problem in scheduling.

4.2.1 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 the real-world schedulers. As evidenced during writing of this report, it is extremely robust. The mechanism of always taking the oldest element naturally acts against high latencies and starvation. The latter, arguably, being one of the most undesirable failure modes in a scheduler. Thus, user can use a FIFO scheduler fairly freely and expect no adverse effects beyond a potential performance penalty. Moreover, additional features tend to be easier to implement (e.g. §4.4.4) as this kind of traversal remains robust even with minor reorderings caused by stealing, overflow structures, incautious workloads, etc.

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 DAG, which hints at its potential for starvation if at least one of the scheduled workloads consist 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 cache. Since memory is hierarchical, the benefits should be noticeable for a number of tasks executed in order and for a wide range of memory requirements.

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

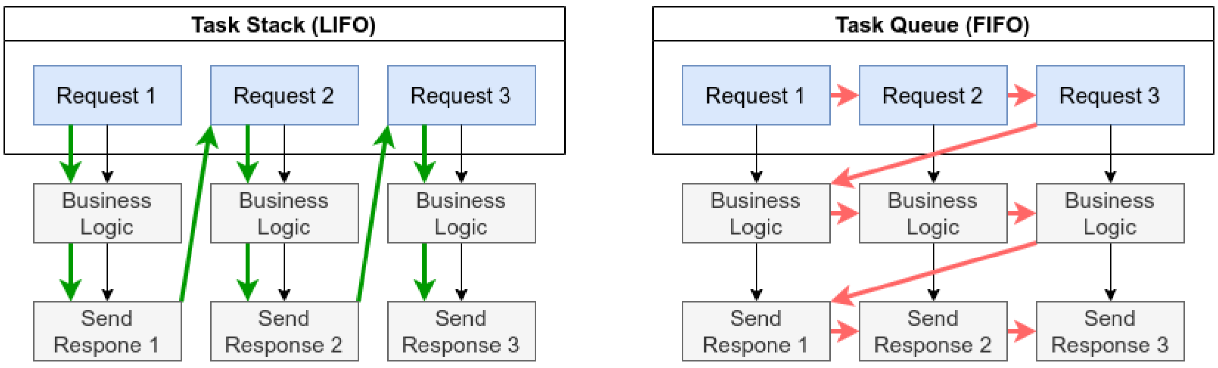

Figure 4.1: The figure shows two ways to traverse tasks graph. Boxes with blue shade represent task physically present in run queue, run stack. The grey boxes constitute tasks that are going to be spawned by their respective parents. To simplify, assume that every task is drawn from the same distribution. In such a case, requests processed using FIFO order terminate at timestamps 7, 8, 9, while using LIFO at 3, 6, 9. Thus, even though FIFO optimizes per-task latency, the latency of actual requests favors LIFO.

Fairness revisited

Previous section (§4.2.1) describes a common understanding of the FIFO and LIFO trade-off. However, as evidenced in §7, choosing LIFO does not need to negatively impact latency. Firstly, if some particular workload benefits significantly from locality, it is likely that both throughput and latency are going to improve. This is especially the case if either stack size is small or tasks move swiftly enough to effectively not spend much time awaiting processing. Secondly, LIFO does in fact prioritise latency, but measured between subtrees of particular tasks rather than tasks alone. Naturally, defining a unit of work to be a task is much more practical and concrete the point of view of software development practice. On the other hand, oftentimes we would actually rather optimize the actual observed latency of e.g. HTTP request rather than the latency between tasks. Thus, in some real-world workloads FIFO-order may actually degrade both key metrics by design. An example of this phenomenon is shown by figure 4.1.

A possible counterpoint to such example would say to simply run the entire request as a single task. Thus making the task-latency and request-latency overlap, what makes FIFO the optimal choice for this metric. However, the workload does not have to be sequential. It might be better expressed as an actual graph of tasks. Splitting the task is also the only way to take advantage of parallelism when system is less loaded. Finally, splitting the request into multiple tasks just lets the scheduler make the decision. This is desired as the scheduler has more information to make the correct call for entire system. Further, a real scheduler is likely far more sophisticated than just FIFO/LIFO and application should not need to make such assumptions.

4.2.2 Hybrid

As discussed in the previous section (4.2.1), 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 heuristic 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, every n elements the scheduler can simply reverse the stack, sacrificing locality in that particular moment to execute overdue work.

could try d
ing adaptal
based on th
oldest elem
in the queu
if have time

4.3 Contention and locality

Previous sections (4.1, 4.2) describe global scheduling properties and mechanisms achieving them. However, as evidenced by , a simple queue (or stack) do not scale beyond a single core. Firstly, imposing thread-safety requires costlier primitives, which are then slower even in the single-core case. Secondly, actual insert/remove operation, which sets in pointer in the array, is significantly faster than fastest atomic operations requiring architecture lock or write buffer flushes. Thus, as the actual parallelizable portion of the work is small, in line with Amdahl's law [29], little performance is gained as number of threads increases.

lifo slot?

queue scala
ity figure

4.3.1 Core Structure Distribution

One way to fix the contention is through having multiple instances of the underlying structure and load balancing across them. In the case of FIFO, the multi-queue [38]. Such a distributed structure is no longer strictly FIFO. But, looking at all pairs of processed elements, the number of reorderings from the precise order is small, and enforcement of a hard bound on reorderings possible [31].

Now, let's focus on multi-queue. There is no need to 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, as single-producer queue requires weaker synchronization than a multi-consumer one (). The core properties of a structure are still maintained by the load balancing on the dequeue side.

ref to imple
mentation

There is one more natural optimization in sight (). Namely, if worker can take an item

ref to back-
ground

from multiple queues, it should prefer to run the work it has scheduled, as it is more likely to benefit from locality and finish faster. An implementation of such a bias is trivial, as worker simply needs prefer to take an element from its insertion queue. Such a local deque operation has a lower constant factor because it does not need to synchronize with insertion. Thus, 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. Similar optimization process applies to LIFO.

To summarize, sacrifice of the global order property for local, per-core structures aims to decrease contention and increase locality. While any deviation from global order acts to increase tail latencies, in this case, it may be balanced off by overall increase of throughput and tasks being finished sooner. Naturally, certain formal guarantees have been given up and that may become more pronounced under specific workloads and during system degradation. The next section discusses an alternative, more stable approach (§4.3.2).

4.3.2 Staged

Staged architecture [47] constitutes a different realization of FIFO order. Different sets of system threads are assigned to handling particular kinds of logic. Those sets are joined with 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**. Task are local to global state of the particular stage. It is beneficial for data-intensive applications (as opposed to compute-intensive [32]) as most I/O operations require use and modification of global state at least at some stage. Here, the thread responsible for setting up TCP connection is able to maintain the list of TCP 4-tuples high in the cache hierarchy. In constrast with e.g. task-locality in FIFO, locality to global state is not going to perish as queue sizes grow.

- **Graceful degradation**. The scheduling architectures with a single structure per CPU described in the previous section (4.3.1) tend to degrade more unpredictably. Work-sharing decreases when most threads become overwhelmed and focus on local queues (for much-needed throughput), what leads to higher variance in their backlogs and thus 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 is no adverse effects on the execution model that compound the issue [47].

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. They can increase the number of servers as load grows but 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 first principles. While such an approach would yield the biggest performance benefits for some workloads, it is also the most constraining one. Throughout this thesis, staged architecture is instead used as a useful model for composing different schedulers. In this case, the main benefits of staged architecture are still reaped but tasks may still benefit from e.g. LIFO scheduling for small tasks within a single pool.

4.4 Other design choices

4.4.1 Work distribution

A widely used specification of the algorithm described in the previous section (4.3) is work stealing. Work-stealing assumes a thread working on the tasks in its local structure as long as there are any. Then, it randomly select another queue and steal half of its items. It has proven to be an effective and robust approach in the past. Nevertheless, this report argues that it is not enough for the present and future.

There are two cases in which work stealing helps a non-preemptive scheduler. Firstly, if only some cores spawn the work (e.g. single web-server), stealing helps to distribute the work among many cores. Secondly, it helps to reduce latency. In my practise of working with Async/Lwt, it is not uncommon for some tasks occupying CPU for much longer than they should be due to a developer's mistake or some IO or database query suddenly taking far longer than it used to, due to hitting scalability ceiling. With a work stealing scheduler in OCaml Multicore, the tasks stuck in the local queue of overtaken core are going to be rescued by other threads as long as they are not overly busy too.

The latter (latency reduction) case is difficult to improve upon. A thread simply cannot know when it is going to get blocked by a lengthy operation. Therefore, unless each thread does additional bookkeeping in some centralized structures for every task, the unoccupied threads have to conduct an active search for lingering work. Naturally, incurring a constant bookkeeping for every task is troublesome to justify, especially for a feature that is most important when overall system is not heavily loaded. Thus, a random search conducted by work stealing is, arguably, a reasonable mechanism to soften the impact of extensive blocking on latency. If predictable latency is necessary, the staged architecture designed in previous section might be a better choice (§4.3.2).

On the other hand, this same mechanism becomes harder to accept for regular work distribution case. Any increase in number of threads is balanced off with lowered chances of finding work created by the same number of spawners. Thus, software engineer needs to be increasingly cautious to achieve high utilization. Even more importantly, in this case the information is available and high-spawner can issue a call for help or simply offload accumulated tasks into a common data structure. Such mechanisms are called *work-shedding*. In this report, the following work-shedding methods have been tested.

o

- dump into overflow queue
- request random thread to steal **ref ot paper from the e-mail**
- dump a ticket into an mpmc queue **need to be implemented**

dump into na mpmc queue is better than forcing a victimg because it will not act on another busy thread.

he case of
ged archi-
ure this is
a problem.

4.4.2 Resizing

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

to back-
und

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

e graph of
-creation
e?

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

ld still try
horizontal
g

4.4.3 Blocking operations

As has been a theme multiple times (), a developed 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 (**ref to tokio**). 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) tries to use the fast, non-blocking interfaces like io_uring.

cite

4.4.4 Yield

Yield operation allows fiber to request descheduling. Commonly, a well-behaved program is expected to explicitely call yield to leave CPU to avoid wasting resources while waiting for an event. However, if the event is a result of computation done by other thread, yield becomes a strictly necessary part of the logic, which allows both threads to terminate. Thus, the design decisions behind this small part of the mechanism have a profound effect on the usefulness of the entire scheduler. As described below, they are also not trivial.

Firstly, pure LIFO-based scheduler is not practical as the underlying data structure prevents implementation of yield operation. LIFO can only operate on the top of the stack and thus yielded task is the only one that can be picked up afterwards. Thus, LIFO ordering needs to be relaxable when needed. Heuristics such as firstly popping the element and then pushing yielded task tend do not fix the underlying issue but increase the number of blocked threads required to deadlock. A more elaborate approach swaps yielded task with a random item in the stack and runs it instead (as shown in section). That fixes the underlying issues but overall performance deteriorates if most tasks in the stack are blocked, and adding swap operation to complex lock-free stack implementations may not be possible. Further, a naive work stealing scheduler may still deadlock. Work-stealing mechanisms tend to bias strongly towards use of local, per-processor queues and steal only if necessary. Such construction improves locality of workload, limits contention and superfluous passing of tasks between processors. However, processor of such a scheduler 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 is periodically checked by processor. Such an approach avoids described issues at a cost of small extra cost in handling yielded tasks, which is acceptable because yielded tasks are unlikely to be latency sensitive or constitute majority of the workload. If that was not the case, either the scheduler can decide whether 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.

ref

Notably, Linux kernel developers have made a choice in a similar vein in 2003 when yielded tasks were changed to enqueue to expired queue rather than run queue (https://lwn.net/Arti

could be
smarter ab
yielding an
use this in

4.4.5 Promise

Promise is a placeholder value returned by a scheduler for a newly added task. It is similar to an option type, which scheduler fills with the value returned by respective task upon completion. If another thread attempts to read the promise when empty, it becomes blocked until the result is known [45].

Fundamentally, promises do not have to be implemented in the runtime. They are a convenient abstraction, that can be built from standard synchronisation primitives as it is in Go . Nonetheless, including them with the scheduler built in this report encourages good habits in the users and allowed audio mixer 2.4.2 and option pricing 2.4.3 benchmarks to be written in a realistic way.

o repo

e listing

4.4.6 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 (2.4.1), 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 [11] or Async's pipe implementation [6].

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 [6].

4.4.7 Cancellations

Important as cancellations are common in modern workloads with requests with deadlines. Further, cancellations are critical to recovering from overloading and thus their implementation has to be particularly robust, as a large wave of propagating cancellations may be the actual event reaching breaking point and causing system meltdown.

Paper a discusses this explicitely. In this work,

batching
would be
faster but i
generally b
for us!: BQ
A Lock-Fre
Queue with
Batching

Chapter 5

Implementation

5.1 Schedulers

to the
ue/stack

As described in §4.2.1, there is an inherent trade-off in scheduling between LIFO and FIFO. Therefore, these two mechanisms have been implemented for foundational schedulers using structures described in . Their implementation follow a scheme with local data structures (§4.3.1) and rely on work stealing for work distribution (§4.4.1) in the base case. On top of these two core structures, the project features a resizable FIFO scheduler, which can automatically scale to required workload, and a staged architecture. The staged architecture is not implemented from first principles. Instead, it utilizes the underlying schedulers as micropools and provides scalable means for scheduling tasks across different micropools. With this choice, it is less efficient but still brings its core benefits and provides another way to compose schedulers (§4.3.2).

Pure LIFO scheduling is prone to poor tail latencies. Therefore, a number of heuristic has been implemented. They aim to integrate FIFO and LIFO ordering in a way that improves upon both, or at least provides a new point in the trade-off space.

- **Alternating**. LIFO and FIFO removal alternately. Simple combination of pro-latency and pro-locality choices.
- **Random**. 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.
- **Temporary FIFO ordering**. 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. Such approach is commonly known as reversing the stack 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. This 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 slightly more costly to use.

Finally, work stealing does not provide optimal performance for highly concurrent workloads [**?**]. Thus, a number of extensions to aid work distribution have been evaluated. They are described in the next section.

5.1.1 Work distribution extensions

As robust as work stealing is, it does have its issues, e.g. blind victim selection, the scalability ceiling describes in the previous section (§7.1.1). As described in §4.4.1, regardles of primary work distribution mechanism, some sort of work stealing is necessary to prevent starvation in non-preemptive design with local structure. Otherwise, with a purely cooperative scheduling, any long tasks is deemed to significantly increase latency of all awaiting items. Therefore, a number of 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 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. This could lead to starvation on blocking, thus the implementation simply has a bias to steal small number of items rarely.
- **Blindness.** Random exploration of other threads local structures is inefficient. It causes contention and suboptimal steals. Instead, 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 threads array. That creates a hierarchy along which tasks move and prevents the underloaded threads at the bottom of hierarchy from contending in the hotspots. Implemented with nearest 15 and 20 threads.
- Do not steal less than 10 elements as they will be efficiently processed by owner. Instead, look for work elsewhere.
- Naive work-requesting. A polar opposite of work stealing. Busy thread randomly chooses another thread, which is forced to steal some of the busy thread's load.

Implemented as non-sticky, only a single steal is forced, and sticky, in which the victim keeps the target until another thread overrides it. Both cases have highly negative impact on this workload. The single spawner becomes a victim of too many threads for efficient stealing. Further, the actual steal happens only once designated thread runs out of work, what may only happen once the steal is no longer needed.

- Work-requesting from busy to free. Requestor publishes its id in a highly-scalable lock-free queue. Threads looking for work picks up the id from the queue and steals work. This has the benefit of closing the information gap of pure work stealing or naive work-requesting. Only the overloaded and underloaded threads participate, while others remain unaffected.
- Overflow queues. Use extra structure for work distribution and put some tasks into a global queue. Here, spawner thread has a chance of inserting a tasks into global structure. Both Tokio (Rust) and Go (`https://github.com/golang/go/blob/master/src/runtime/proc.go#L2597`) feature global, locked queues for any tasks overflowing local queue. Three queues have been implemented: locked, lock-free, lock-free, 5-multi-queue. **what effect**
- Local resizing. Stealers aims to take half of its victims tasks. Increasing the size of local structure and the number of awaiting itmes may in turn lead to higher number of items distributed per steal.

5.2 Underlying Data Structures

This section describes the implemented schedulers and, briefly, underlying lock-free structures.

5.2.1 Core Data Structure

Lock-free data structures tend to be difficult to use in the real-world. Firstly, all assumptions about their naive counterparts are off once fine-grained concurrency is in place. For example, such a simple method as size() often turns out to be non-trivial to make linearizable 2.3.4 when size of a linked-list or circular queue changes in multiple places at once. This makes the trade-off space far more complex and points in that space (algorithms) more varied than in the single-thread world.

Secondly, the effort required to add new features increases as a function of current complexity. With a single-threaded code, developer implementing a new function can focus solely on the state before and afters its execution. That alone is enough to examine whether the new function fits with 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. Thus, such function is correct as long as its internals follow requirement. This could not be further from the truth with lock-free concurrency. In fact, a new function has to be made safe to execute concurrently with entirety of existing codebase. Even if the function is trivial, it may render some seemingly unrelated portion of the code non-linearizable.

Therefore, this thesis relies on a lock-free structure inspired by **?**]. The structure is described in detail in the next section. Before that, it is worth touching on the main advantages of such approach.

- **Scalability**. CAS and FAD operations take approximately the same time **PAPER B**, but traditional CAS-based queues scale poorly due to number of CAS fails growing proportionally with number of threads. Here, despite additional CAS required, all the significant contention is resolved by FAD (**See result X**.) and thus, this algorithm scales similarly to state-of-the-art FAD queues.
- **Simplicity**. The base case of this structure takes a little over 10 lines of lines of code. It is unusually concise for a practical multi-producer multi-consumer LF queue.
- **Flexibility**. As discussed, lock-free implementation can be unwieldy. This posed a particular threat to this thesis, as ultimately, the experiments should aim to evaluate scheduling methodologies rather than just show which underlying data structure can be implemented in the fastest way on existing architecture. CAS+FAD queue has the benefit of being extremely flexible for a LF data structure and was adapted to support LIFO, non-blocking returns on full or empty and single-producer scenarios with relative ease.

describe fas
local queue

5.2.2 MPMC Queue Blocking

5.2.3 Relaxation to SPMC Queue

5.2.4 SPMC Queue with stealing

5.2.5 SPMC Stack with FIFO stealing

5.2.6 Sharded Queue

5.2.7 Overflow avoidance

5.2.8 False sharing

5.2.9 Limitations

Since this project has a limited timescale, the schedulers were optimized in design (for e.g. scalability) but their implementations were not optimized

Chapter 6

Methods

This section describes details of the testing experimental setup: hardware, system and testing procedures. Afterwards, it justifies the statistical methods and discusses probe effect.

perf stat

6.1 Hardware

6.2 System

mention thread limit in ocaml

6.3 Procedures

reject first sample lazy loading and caches

how many times executed

performance sensitive data was collected on thread-local counters

6.4 Statistical Methods

look for extra variance use medians and quartiles as its h

6.5 Probe Effect

Story about vdso.

Chapter 7

Evaluation

This chapter presents 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. This result is analyzed in depth (§7.1.1) and a number of extensions favoring fine-grained concurrency are evaluated (§7.1.1). Further, the section focuses on more complex workloads. First, web-server with cache effects (§7.1.2) and bimodal task processing time (§7.1.3). Second, the audio mixer benchmark.

7.1 Server benchmark

7.1.1 General Case

The first benchmark mimics a minimal TCP packet processing scenario. The workload relies on a single spawner as it would be the case with a server listening on IP and port tuple. To minimise the noise, actual network is not used. Instead, the *server* thread chooses one out of 10 pre-made packets, copies it and schedules a handler task by performing an appropriate effect. The benchmark focuses on max load scenario and spawns tasks as fast as only possible, until the requested total is reached. Then, 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 spawner never removes elements from its local structure. Local structure sizes are held constant at 128 elements. ()

ref to justif
queue size
- generally,
don't want
too much
waiting bc
memory pr
sure, fiber
recycling,

The results of conducting a test with 50k packets in such a controlled setting are show by figure 7.1. The first panel shows mean processing speed on a logarithmic scale. In the first part of the graph, between 5 and 20 domains, the performance rapidly accelerates with increase in the number of threads. 20 domains witness peak performance by LIFO scheduler and as more threads are added, the performance degrades. At first slowly, then

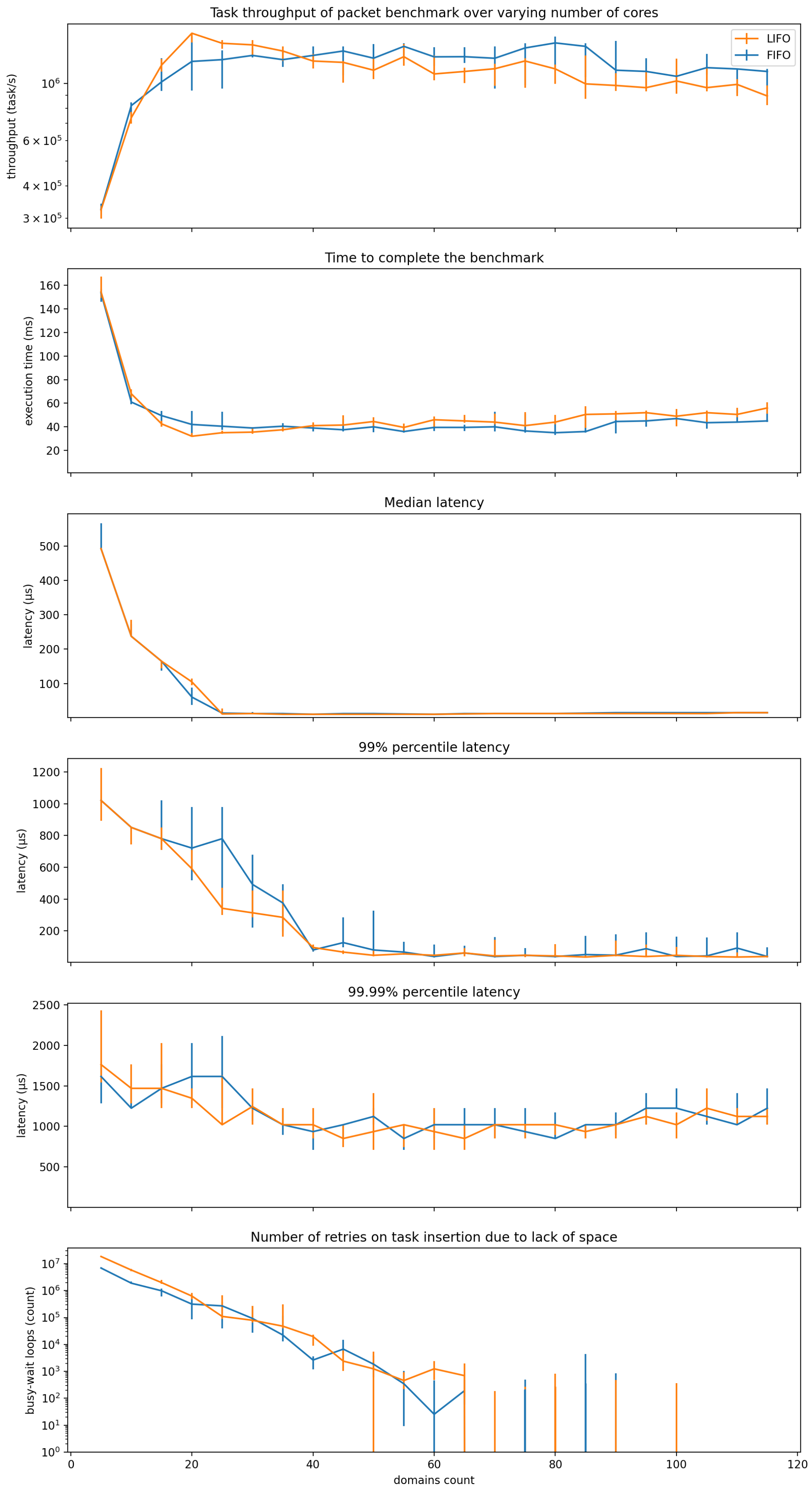

Figure 7.1: 50k.

faster 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 [29] but it should not be detrimental either. Discussed further in §7.1.1.

The second panel shows actual time required to complete the benchmark. Next, the charts from three to five show 50%, 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 FIFO offering a better median latency for one point and LIFO better tail latencies for 2-3 points. Afterwards, the results converge. The tail latencies are partly driven by LIFO simply 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 number of retries of the main spawner (*listener*) when trying to insert elements into the queue or stack. The retry loop is relatively cheap to execute . Clearly, for lower core counts the spawner frequently throttles adding tasks due to lack of space. However, the number of busy-waiting cycles lowers with extra cores, quickly reaching point where magnitudes are too small to affect overall performance. At last, it ceases to occur in the median case beyond 45 cores.

link to the code in the repo

Notably, both FIFO and LIFO perform very similarly on all metrics and exhibit similar phenomenons. As hinted in the first paragraph, the workload is so simple that it does not let some of the individual advantages prevail. There are slight differences, i.e. LIFO peaks at 20 domains, while FIFO at 80. But their overall performances (throughput and latencies) in ranges 0-20, 20-80, 80-115 are comparable.

Overall, it shows that both implementation are fairly balanced and a good foundation for benchmarking more complicated workloads.

Degradation

In line with Amdahl's law [29], it is generally expected that an application may not perform better with more resources. After all, scalability is something to explicitely design for and even then, may not always be possible . However, there is no justification for more workers actually causing harm to the performance. Such a characteristic is highly undesirable in such a foundational system as scheduler. However, why would it arise in such a battle-tested idea as work stealing?

ref to hash chain exam

In the LIFO case, as shown by 7.2, the scheduler is unable to maintain the increase in stealing attempts. Such 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, the other thread has 100% hit rate. On the other hand, if it is 1 out of 10 threads, then there are are likely going to be some misses or inefficiencies (minimal amount of work acquired). The diagram shows, that above 80 domains, as the information becomes more

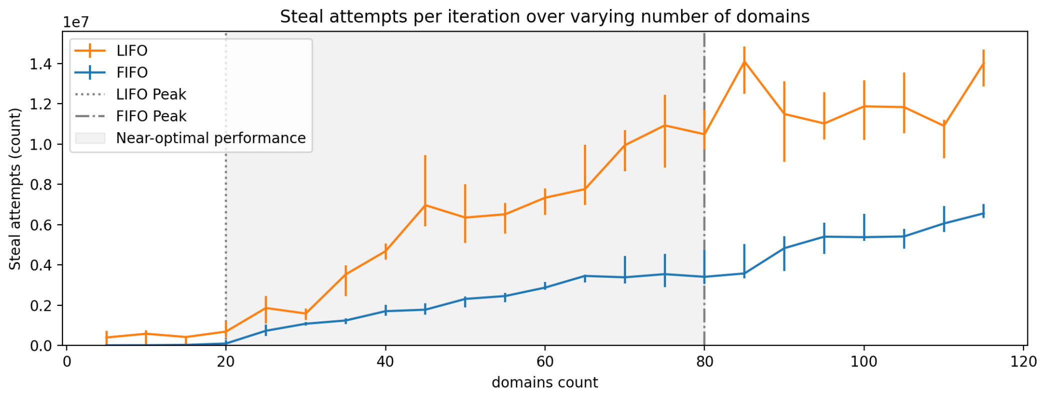

Figure 7.2: The throughput achieved on 50k tasks over varying number of domains.

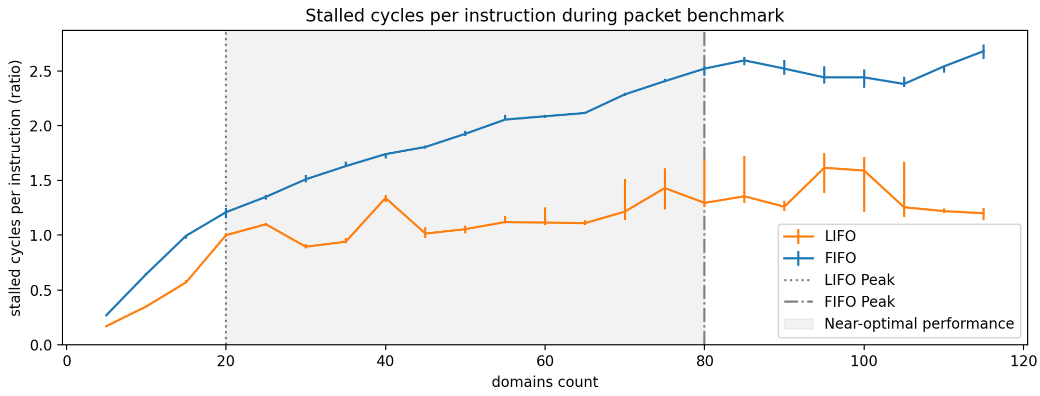

Figure 7.3: The throughput achieved on 50k tasks over varying number of domains.

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 but in practise there is only so much cross-core coordination the processor can handle . This effect compounds, since the starved cores keep retrying, thus impeding the actual useful attempts. Conversely, if there is significantly more work and more time, the tasks saturate the system enough to prevent excessive stealing and observed negative effect does not occur (7.4).

SI link

ink this agraph

In the FIFO case, the problem also relates to work distribution. Generally, 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 . Therefore, while stealing alone is not maxed out, it significantly affects the rest of the logic. As shown by figure 7.3, 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.

ld show agram of f tasks cessed by n core.

o imple- tation

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 its own logic and requires allocations but it does constitute a very fine-grained concurrency. That puts a high pressure 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 due to e.g. atomic operations taking constant time and load information

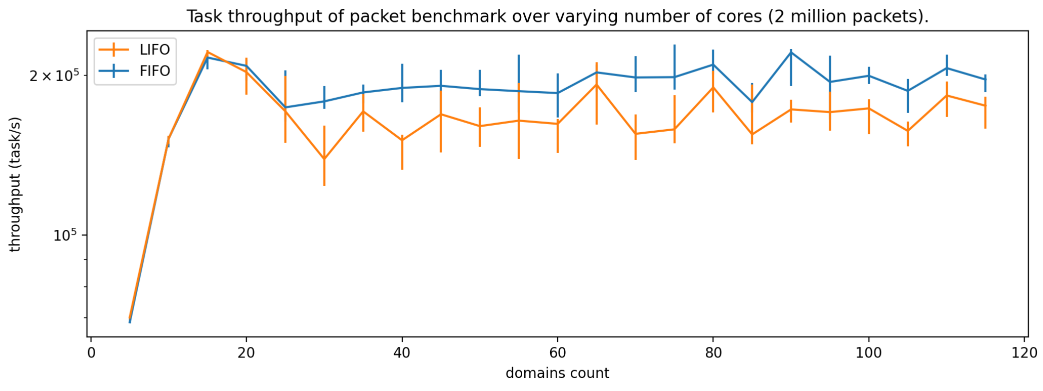

Figure 7.4: The throughput achieved on 10 million tasks over varying number of domains.

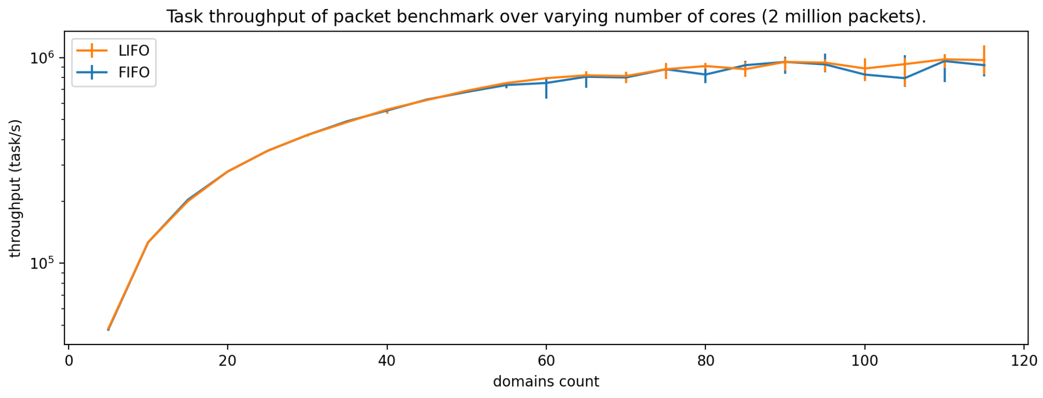

Figure 7.5: The throughput achieved on 50k tasks with over varying number of domains. Each tasks loaded with additional 500ns wait.

is being less dispersed. If the workload is simplified just a little, the scalability improves.

- **Heavier task**. See figure 7.5 for the performance when each task has had an extra 500ns sleep to perform. This is enough time to extend the stability beyond 100 domains.

- **Higher workload**. In fact, increasing the number of tasks also stabilizes the scaling. See figure 7.4 for the case with 10 million task instead of 50k. This case saturates many more workers, and while overall throughput is lower, the performance overall

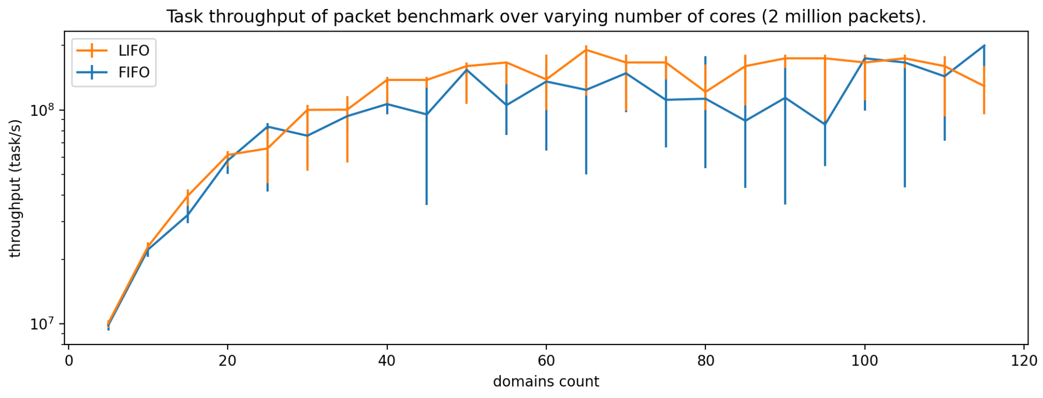

Figure 7.6: The throughput achieved on 50k with 4 spawners. (2 spawners in the 5-core case.

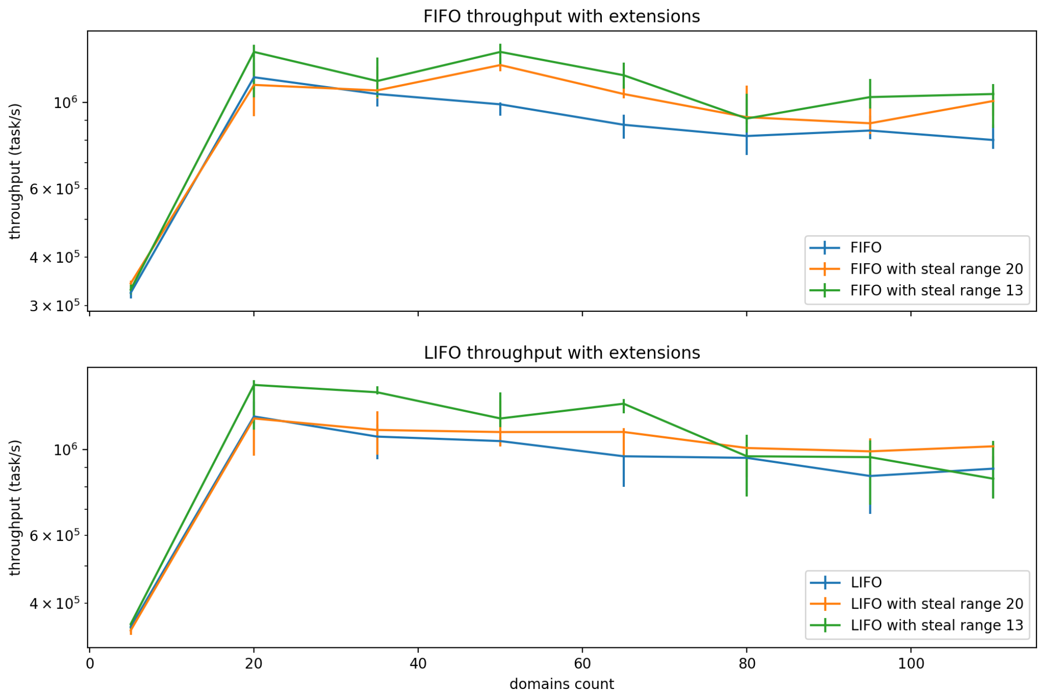

Figure 7.7: The throughput achieved on 50k tasks with over varying number of domains. Each tasks loaded with additional 500ns wait.

long-
ning,
ch is un-
al in e.g.
traffic

remains stable beyond 30 cores.

- **More spawners**. See figure 7.6 for the performance when there is more spawners, i.e. developer made an explicit change to help the scheduler.

Extensions

§5.1.1 discussed a number of alternative work distribution mechanisms. The positive results are shown in 7.7. Only neighbourhood stealing has shown a clear improvement over the baseline, which is in line with the diagnosis stated in the previous section §7.1.1. It is worth pointing out that such an optimization is going to have a negative effect on coarse-grained workload, where threads higher in the hierarchy are going to be stuck processing tasks for longer and thus not replenish its local structure regularly.

Only stealing at least 10 elements had a strong negative effect, as it paradoxically increased the number of underloaded threads and contention on the loaded ones. Similarly, naive work-requesting approaches resulted in other threads overloading the spawner and causing less efficient work distribution than random work-stealing, where 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 has to compete with the workers for the lock.

Finally, scalable overflow structures (lock-free queue, multi-queue) and advanced work-requesting had an insignifcant positive effect. They would likely yield a stronger benefit for less heterogenous workload. Likewise, local resizing would be beneficial if starting size

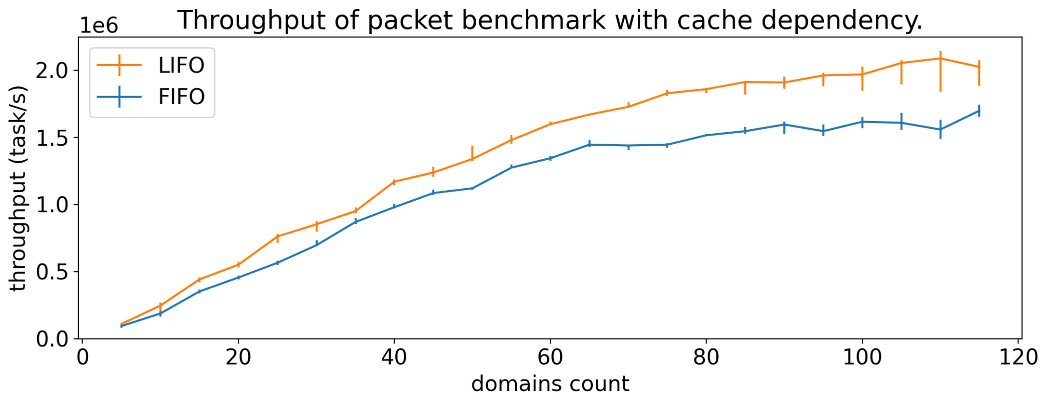

Figure 7.8: The throughput achieved on packet processing benchmark with 500k packets and 10 spawners. The benchmark was modified to have a non-trivial cache effect. Once the task was processed as in the original benchmark, it would spawn a chain of 10 tasks. Each task reads 20 with each reading 20 random bytes from the packet buffer, what simulates picking out different parts of the received data at different stages. LIFO offers a very clear throughput advantage (shown in the figure) and median-tail latency improvement.

of the local structure was small enough.

7.1.2 Server benchmark with cache effects

This section presents 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 (§7.1.1) 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's until the depth of 10 is reached. Moreover, there are 5 spawners instead of 1, and individual queue sizes have been increased to 1024 to accommodate that. The results are shown by figure 7.8. Such a workload naturally favours LIFO, which can make a much better use of hierarchical memory and reduce cache misses. Thus, achieved throughput is far superior with LIFO ordering. Further, in line with the mechanics described in §4.2.1, 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 consecutive task does not experience memory inefficiencies due automatic memory prefetching. After a few first cache fails, the rest of the packet is going to be loaded into cache preemptively. This effect breaks down with any more complex memory patterns, which microarchitecture cannot predict.

7.1.3 Server benchmark with multi-modal processing latency

Figure 7.9 shows results of an alternative twist. Here, instead of spawning 10-sequence, only 3-sequence is spawned and the last task has a 0.1% chance to spawn another task with lengthy operation. The 3-sequence still has cache-dependcies, which favour LIFO, but optional final task sets a trap. If scheduler processes the lengthy operations without

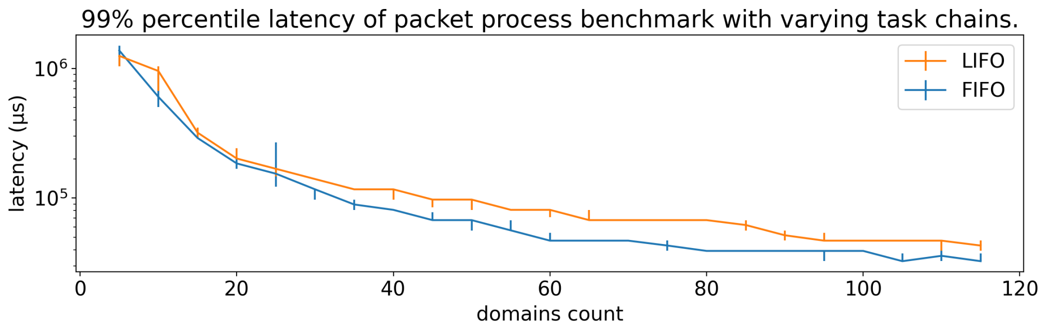

Figure 7.9: The throughput achieved on packet processing benchmark with 100k packets and 10 spawners. The benchmark has a weaker cache effect than 7.8 but still significant enough for LIFO to have a better throughput. However, there is a 0.1% chance that packet causes heavy work to be done, which spawns two tasks taking 20ms to complete. Akin to a network call to a different service. As shown in the figure, FIFO clearly 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.

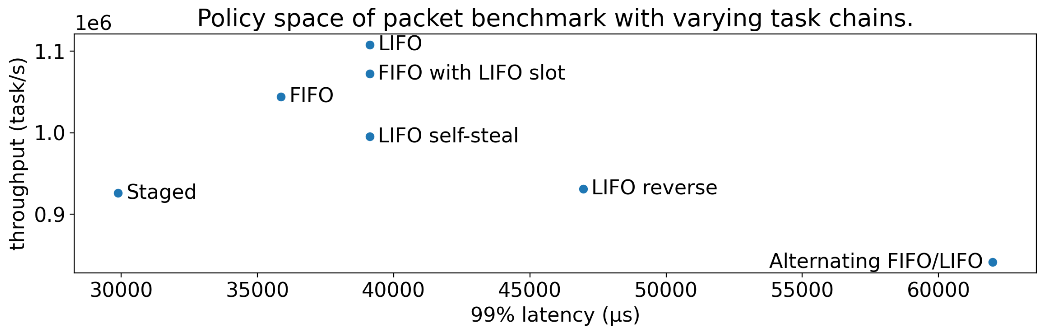

Figure 7.10: todo!!!.

care, it risks massively increasing latency of all awaiting tasks. This favour FIFO, which is able to process lightweight sequences concurrenctly 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 minimal impact.

alternative explanation: that strongly favours FIFO. Notably, the workload is still locality-sensitive but now, it also features a variance in depth of task-graphs spawned by packets and cost of processing individual tasks. Suddenly, FIFO is able to offer a significantly better tail latency as it actively seeks to process the oldest tasks. Conversely, LIFO lets lightweight tasks remain stuck behind random heavy jobs.

Other schedulers

Finally, a number of other approaches described in §5.1 has been tested. See figure 7.10. 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 processing the lightweight tasks. The fall in throughput is difficult to avoid in a workload that work stealing is able to scale efficiently. Nonetheless, if the final task was lighter, it is possible for staged architecture to improve on both metrics, as observed with artifical benchmarks.

7.2 Mixer benchmark

yielding, promises, overflow queue

very concurrent - queue size important

Chapter 8

Summary and conclusions

As you might imagine: summarizes the dissertation, and draws any conclusions. Depending on the length of your work, and how well you write, you may not need a summary here.

You will generally want to draw some conclusions, and point to potential future work.

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] Ocaml library : List. https://v2.ocaml.org/api/List.html. (Accessed on 05/18/2022).

[5] Power4 - wikipedia. https://en.wikipedia.org/wiki/POWER4#cite_ref-stallings_2-0. (Accessed on 05/17/2022).

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

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

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

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

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

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

[12] 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.

[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] 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.

[21] 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`.

[22] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark sil-

icon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[23] 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`.

[24] 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`.

[25] 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`.

[26] 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`.

[27] 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`.

[28] 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`.

[29] 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`.

[30] 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`.

[31] 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 Heidel-

berg, 2013. doi: 10.1007/978-3-642-39958-9_18. URL https://doi.org/10.1007/978-3-642-39958-9_18.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIG-PLAN International Conference on Programming Language Design and Implemen-*

tation. ACM, June 2021. doi: 10.1145/3453483.3454039. URL `https://doi.org/10.1145/3453483.3454039`.

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

[44] 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`.

[45] 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`.

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

[47] 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`.

[48] 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`.

[49] 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

Technical details, proofs, etc.

A.1 Lorem ipsum

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam