An OS-level adaptive thread pool scheme for I/O-heavy workloads

by

Jannes Timm

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday February 25, 2021 at 9:30 AM.

Student number:	4907191	
Project duration:	May 1, 2020 – February 25, 2021	
Thesis committee:	Dr. J.S. Rellermeyer,	TU Delft, supervisor
	Prof. dr. ir. D.H.J. Epema,	TU Delft
	Dr. A. Katsifodimos,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

I would like to thank my supervisor Jan for his constant support throughout the whole duration of this thesis project. The weekly meetings and the freedom to research according to my ideas and interests made for an enjoyable experience. I am also grateful to Sobhan, who joined part of the weekly meetings and who also provided assistance personally at some points in the project. Lastly, thanks to Dick and Asterios for being part of my thesis committee, and thanks to friends and family who also supported me throughout the period I was working on the thesis.

Jannes Timm
Delft, February 2021

Contents

1 Introduction

With increase of CPU cores and higher demands with regards to scalability, concurrent programming and application architecture is becoming more prevalent in the software world. Even before this trend the C10k problem illustrated that the memory, creation and scheduling overheads of operating system (OS) threads prohibit their excessive usage, i.e the thread-per-request model commonly used in web servers was not scalable enough for the increasing amount of concurrent requests [16]. A solution to the problem of overheads of too many OS threads and their repeated creation/destruction is the pooling and reuse through the introduction of thread pools within an application [25].

Thread pools alleviate the overheads of dynamically creating and destroying threads per request, but it is often not clear what their optimal size (amount of threads) should be. Different schemes for dynamic adjustment of the amount of threads have been suggested and implemented. Most of these assume general workloads, while in many applications and libraries thread pools are employed for I/O-bound workloads only, especially disk I/O [1]. Many modern runtime libraries that enable asynchronous programming architectures, such as Tokio [30], Libuv [26] and ZIO [31], use a thread pool solely for disk I/O and disk related blocking operations. A cause for this restricted use case has been lacking or immature asynchronous I/O interfaces for disk I/O, whereas network I/O has better support for asynchronous programming.

Common adaptive strategies that assume a generic workload fail to capture throughput limits of local disk storage, resulting in the excessive creation of threads which wastes memory and may affect throughput negatively. With the restriction of the workloads to disk I/O jobs only we are able to correlate observable metrics such as logical disk throughput to the general throughput of the thread pool, which enables a feedback-based algorithmic approach that relies on realtime OS metrics.

1.1. Research Objectives

In this work we focus on developing an adaptive scheme for a specialized thread pool, that solely executes jobs that mostly perform input/output (I/O) on local disk devices. Adaptive here means online adaption of the amount of threads in order to increase performance. We aim to maximize the throughput while minimizing the amount of worker threads in the thread pool. Specifically we target the Linux operating system as it is easily extendable, and we restrict ourselves to disk I/O because in asynchronous application architectures network I/O is already often implemented in a non-blocking, poll-based model, that make the use of a thread pool redundant, whereas disk I/O is commonly delegated to a thread pool. We formulate the following research questions, which are addressed in this work:

1. Which OS metrics are useful in characterizing the performance of a disk I/O thread pool?
2. What is an effective approach to dynamically adapt thread pool sizes in order to increase throughput while minimizing amount of used threads?
   (a) Is the overhead of the dynamic resizing significant?

[1] we use term ”disk I/O” to mean I/O operations for all types of storage devices, in contrast to network I/O

(b) Can we achieve the same throughput as a fixed-size pool with optimal size for workloads with homogeneous jobs?

(c) How does the performance w.r.t to throughput and amount of used threads compare to the Watermark scheme [2]?

(d) How does our approach perform for more dynamic workloads with heterogeneous workloads and changing jobs submission rate?

Question 1 and 2.a are addressed in chapter 4 when we introduce our solution approach and the remaining questions are answered in chapter 5 and 6 through experimental analysis.

Furthermore w.r.t to the implementation of our adaptive scheme we propose the following goals:

- Minimize the amount of configuration parameters, the approach should perform acceptable for all possible disk I/O workloads when using default values (it is unavoidable that for different workloads different configurations achieve best performance, but as we are proposing an adaptive scheme to avoid having to tune a fixed-size pool's size, we feel it is an important design goal)
- Achieve throughput close to the optimal fixed-size pool for stable workloads with homogeneous jobs (we expect the optimal size of the pool to be stable for such workloads, so there should not be much potential to outperform the best fixed-size pool)
- Outperform the optimal fixed-size pool for workloads with stable phases of different load/jobs (when a workload consists of multiple stable phases, it is likely that the optimal pool size is not the same over the whole workload, so an adaptive scheme should be able to achieve higher throughput than the optimal fixed-size pool here)

The solution approach we opt for takes advantage of the restriction to disk I/O only jobs through maximizing logical disk throughput, which we show to be highly correlated with the total runtime for such restricted disk I/O workloads. We compare our solution to the optimally sized fixed-size pool and the commonly implemented Watermark scheme (e.g Java's ThreadPoolExecutor [22] and MariaDB's thread pool [28]).

We believe the main contributions of this thesis are the following; firstly, we show that using OS metrics for determining concurrency levels for workloads consisting of jobs that mainly consume a certain type of resource is a sensible approach that can minimize the amount of threads used while achieving good throughput. Secondly, we develop an adaptive thread pool implementation based on live OS metrics as feedback, discuss its components and architecture, and show that the introduced overheads by the controller are negligible. Furthermore, we compare the performance of our proposed solution to fixed-size pools and the Watermark scheme, showing similar or better performance for a variety of read-write workloads. We also integrate the main component of our solution, the scale controller, into 2 existing applications, Node.js and RocksDB, making their internally used thread pools adaptive, and experimentally evaluate the performance compared to default and tuned setups.

1.2. Outline

In chapter 2 we shortly introduce the reader to some related background about I/O interfaces, threading models and observability in the context of the Linux OS. In chapter 3 previous research on adaptive thread pools is discussed, while contrasting methodology and use cases with this work. The following chapters develop and experimentally verify our solution. Chapter 4 contains an analysis on OS metrics that are correlated with thread pool job completion throughput, a high-level description of the algorithmic approach and the architecture of the complete proposed adaptive thread pool. In chapter 5 we introduce a model for multi-phase workloads and evaluate our solution experimentally against a set of synthetic and application workloads. Our implementation is integrated into two widely-used project, RocksDB and Node.js, and benchmarked against the default thread pools in these projects. In chapter 6 we refine the approach to deliver better results for the tested Node and RocksDB workloads and we discuss more generally the implications for other applications. We conclude this thesis in chapter 7, summarizing the results and contributions, and finally briefly discuss possible future work.

[2] the Watermark scheme keeps the pool size between a minimum and maximum number of worker threads, scaling up when more jobs are available and terminating workers after some period of being idle

2 Background

2.1. Compute-bound & I/O-bound Workloads

Computing workloads can be categorized w.r.t their resource usage. A workload that predominantly uses the CPU with only accessing data residing in cache or main memory is called compute-bound. If in contrast the workload consists of a lot of interaction with I/O devices such as disk and network, it is called I/O-bound. These two types of workloads are very different in their interaction with the operating system (OS), a compute-bound process usually make full use of the processor time it is given by the OS, whereas an I/O-bound process often yields the processor back through the use of blocking system calls. For this work we are only focusing on I/O-bound workloads and their characteristics in concurrent settings.

2.2. Taxonomy of I/O Access

In the following we discuss and classify different types of input/output (I/O) operations. An operation performs I/O access when it involves transfer of data between memory and an I/O device. The following discussion is limited to interfaces of disk and network devices. In Linux the common abstraction to interact with these devices is the file.

I/O access on modern Operating Systems can be characterized into different categories depending on whether it is synchronous/asynchronous and whether it is blocking/non-blocking. Which access method is used is largely dependent on an application's architecture and the concrete I/O APIs offered by the OS. We introduce Linux-specific I/O interfaces and categorize them according to their access properties.

2.2.1. Synchronous vs. Asynchronous

Synchronous I/O access means an application issues I/O operations and then waits for the results before continuing to do other work. Conversely, asynchronous I/O access is characterized by performing of other work during the period between the issuing of an I/O operation and the completion of it. Figure 2.1 illustrates an example execution of a thread that first performs some system call that synchronously returns a result and then some call that returns a result asynchronously. For synchronous access the thread retrieves the result of the operation right after the system call returns (e.g read system call), thus not executing any further instructions between issuing the operation and retrieving its result. For asynchronous access the thread does not retrieve the result of the operation when the system call returns (e.g asynchronous read through io_uring_enter system call), it then goes on to execute other instructions before retrieving the result at some later point. This is just one example for synchronous and asynchronous access patters, we will discuss specific interfaces in more detail below.

2.2.2. Blocking vs. Non-blocking

An operation, in our context the invocation of a system call, is blocking if it potentially results in suspension of the current thread of execution. The suspended thread is in a blocked state until the operation is completed and the operating system reschedules the thread. Conversely, a non-blocking operation

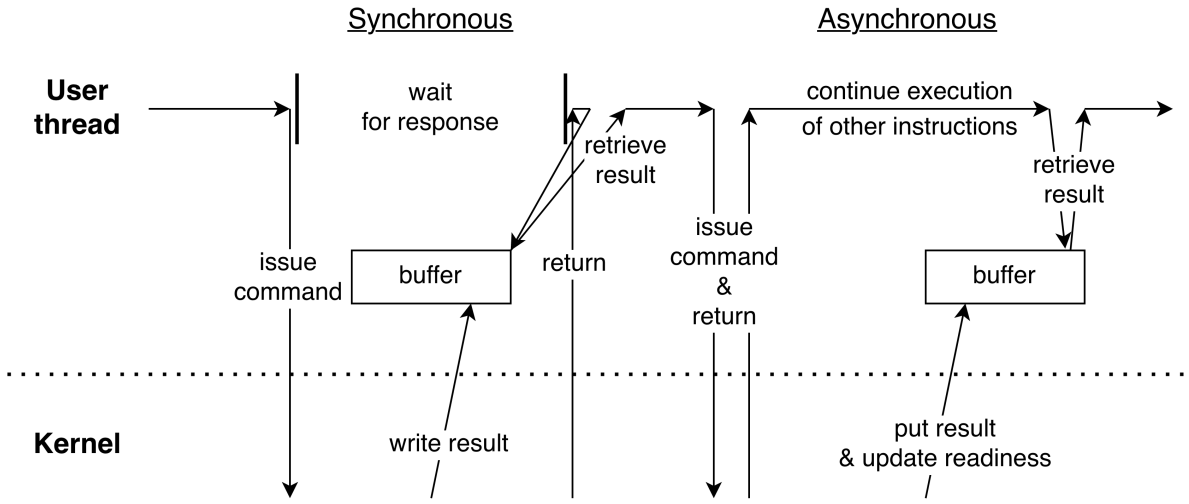


Figure 2.1: Synchronous and asynchronous system calls

is guaranteed to not block and therefore invocation will not result in the invoking thread to be unscheduled. For example in figure 2.1 the synchronous operation may be a read system call that results in blocking the thread because the requested data is not present in cache, or it may be a read system call in non-blocking mode that either returns the requested data if present or return an error if not.

2.3. Linux Kernel I/O Interfaces

Linux offers a variety of system calls and flags that enable different kinds of I/O operations. We offer a short overview of the main system calls for performing I/O and some of their commonly used modes of operation, as well as a classification according to the above taxonomy.

2.3.1. Synchronous, Blocking Read/Write

The more traditional access method is synchronous, e.g the `open`/`write`/`read` system calls which date back to the original Unix system developed in the early 1970s [24]. These system calls are the simplest way of performing I/O work on Linux and most suitably used in non-concurrent applications or generally in synchronous application architectures. By default the `read` call blocks if the requested data is not cached in the Linux page cache, the `write` system call may or may not block depending on whether any read needs to be performed to realize the write operation.

With increasing demands on highly concurrent applications such as web servers the need for asynchronous architectures that share threads of execution arose. To realize an asynchronous execution model one can defer synchronous I/O access to a dedicated thread pool, which does not require new I/O access methods. This approach is very common for performing concurrent disk I/O and all the workloads we use to evaluate our adaptive thread pool solution use these system calls or variations thereof (e.g `pwrite`, `pread`).

2.3.2. File Descriptor Monitoring + Non-blocking Read/Write

The `read`/`write` system calls can be used in non-blocking mode, which instead of blocking in case a file is not ready to be written to/read from just returns an error. This can be used in combination with several methods of monitoring file descriptors for their readiness to implement synchronous and asynchronous non-blocking I/O access.

There are several system calls - `poll`, `select`, `epoll` - that allow for monitoring file descriptors for their ”readiness” to do disk I/O. A file descriptor being ”ready” means that one can perform `read`/`write` system calls which will not block the calling OS thread. So a common pattern for ensuring an application’s I/O operations are non-blocking is to always wait, either busy looping or blocking the thread, for some file descriptor to be ready, then perform I/O on that file or the multiple files that are ready, and then check again for more file descriptors to be ready.

This model is a more complex alternative to deferring synchronous I/O calls such as `read`/`write` to

a thread pool, with the same goal of achieving an asynchronous application architecture. It is commonly used for implementing asynchronous network I/O, but is only partly applicable to disk I/O, as this family of polling system calls does not support disk files.

2.3.3. Memory-mapped I/O

With the `mmap` system call an area of a process' memory can be mapped to a file residing on disk. Then I/O can be directly performed reading/writing from/to that area in memory. Using mmaped file I/O avoids the need for making system calls, and specifically avoids the usual copying of data from user process memory to kernel memory. As this method of I/O access is just reading and writing to memory, it is synchronous just like normal memory access. Depending on whether the underlying file is cached, accessing mmaped memory may or may not be blocking.

2.3.4. Asynchronous Read/Write

While the Linux AIO asynchronous I/O interface has existed for some time, it has severe limitations and has not seen wide adoption [4]. Most notably it is restricted to direct disk access (O_DIRECT flag) which bypasses the kernel page cache. Recently a new interface has been added to the kernel, called io_uring. It offers functionality for fully asynchronous non-blocking disk I/O without the limitations of AIO. Using io_uring applications can submit read/write requests in a non-blocking fashion without having to poll for readiness and consume read/write completions in either non-blocking poll-based or blocking fashion. We'll have more to say about io_uring and how it relates to our work in the coming chapters.

2.4. Concurrency Models - Threading

Concurrent applications have to choose an approach on how to use and organize OS threads to make their architecture concurrent. On an abstract level an application (or application component) has to process requests and produce responses. The simplest threading model is thread-per-request, where a thread is created to handle a request and destroyed after delivering the response. This approach does not scale well for many concurrent requests and furthermore it wastes system resources through the overhead of continuous thread creation and destruction. A thread pool with a fixed amount of threads enables the reuse of threads and thereby avoids creation/destruction overheads. By decoupling the request/response handling from actual processing, which is delegated to the thread pool, an application can scale much beyond the actual amount of worker threads in the pool. As the optimal amount of worker threads is dependent on many factors such as request load, type of requests and the available system resources, the optimal configuration is hard to determine a priori and in most cases the optimal amount of threads will vary during execution.

A commonly implemented and widely used approach that constitutes the next step after a fixed-size thread pool is the ”Watermark” model, which keeps the thread pool size between a minimum and maximum amount of threads based on current load [11, 22]. When a worker thread's idle time exceed a certain threshold it gets destroyed, when all worker threads are busy at the time of a new job arrival and the current amount is below the maximum, new workers get created. This scheme may work for some cases, especially when the jobs are mainly CPU-bound and the maximum size equals the amount of threads that the CPU can execute in parallel. However, for mainly I/O-bound jobs this model fails to capture the utilization of system resources such as the disk. We later experimentally verify this claim and compare this model to our proposed solution.

2.4.1. Asynchronous I/O - Application Architecture

Applications and libraries can opt for different approaches to implement asynchronous I/O. In the following we discuss two common approaches and their implications.

In the poll-based approach blocking I/O is avoided by monitoring file descriptors for readiness. Only when files are ready to be read/written the read and write system calls are issued and as a result they will never block. As the main thread will execute this steps continuously in a loop, this model is called an event loop.

Another approach is to have a dedicated pool of threads which do any potentially blocking I/O operations. This prevents a main thread to ever block at the expense of extra system resources, especially memory, being used up by the threads that perform the blocking I/O.

Even in event loop architectures there are still blocking I/O system calls that have to be performed on an extra thread to avoid blocking the main thread. One example is the `fsync` system call, which is required for flushing the page cache to disk and therefore guaranteeing that a file is durably written to disk. So in practice even loop based systems still have a dedicated thread pool for these kind of blocking calls.

2.5. Linux Observability

There are multiple options for obtaining metrics for a given Linux process. By default Linux already exports basic process statistics through the `/proc` filesystem. These are accessible without root privilege and provide a diverse range of information about a process.

For more detailed and custom (performance) metrics about a process one has to proactively trace a process [12, 13]. This can be achieved with various different tracing tools that make use of kernel built-in tracing mechanisms. Multiple mechanisms such as kernel/user probes, tracepoints and function hooks are built into the kernel and are exposed through various means to user space [27]. Different tools such as systemtap, ftrace and perf use these interfaces in different ways. Recently a push towards unifying the tracing infrastructure aims to develop a common user library that acts as a unified user-space interface to the kernel-space tracing machinery [8].

These tools usually require root access, as they have the capability to trace any running process on the system. So they are not practical to use for applications that want to use elaborate system metrics to adapt their own behavior, such as an adaptive thread pool, as applications are commonly not run as root for security reasons.

2.6. Measuring Performance

Performance may be quantified with different kind of metrics, typically the two most important ones are throughput (requests completed / second) and response time (amount of time it takes from issuing a request until receiving the response). Depending on the application it may be more important to reduce the average response time or specifically reduce the response times of the worst percentile. While generally aiming for a higher throughput it may also be important to keep the throughput at a steady rate instead of a high average throughput that is very volatile over time. For the purpose of evaluation performance of our adaptive thread pool we will use the average throughput, which is analogous to the total runtime of a workload.

3

Related Work

There has been much research on making thread pools adaptive to increase their throughput or decrease latency. Most work focuses on thread pools that have generic workloads, jobs that are submitted to be executed may be I/O-bound, compute-bound or a mix of both. Therefore the used metric for throughput is often also generically defined to be the rate at which submitted jobs are completed and latency is often determined by job completion times.

Previous work can be distinguished with regards to the algorithmic approach, the target metric to be optimized and the use of auxiliary metrics obtained by monitoring or known a priori. Generally either latency or throughput are optimized, with throughput usually being defined by the amount of jobs completed per time. Algorithmic approaches range from manual tuning of thread pool size by system administrators, to the optimization of complex theoretical models that factor in a variety of variables that may not be easily observable [2]. Model-based approaches aim to directly calculate the optimal thread pool size, whereas feedback-based approaches adjust the amount of threads based on a set of observed metrics. Much of the feedback-based algorithms are rather simple, using some easily obtainable OS level metrics or none at all, and adopting linear search or hillclimbing approaches [32].

Ling et al. developed a theoretical model to compute the optimal pool size based on a priori knowledge such as a probability distribution over the request load and the thread creation and context switching costs [21]. Applicability of this model is restricted by the need of a good estimate of the expected load and further assumptions like homogeneous jobs (similar I/O and memory utilization) and constant threading overheads. As the authors note, ”[t]he optimal size of [a] thread pool is a dynamic performance metric that is influenced by the request activity, the system capacity and the system configuration”. In contrast to their work, we make no attempt to model the request (job submission) activity and system capacity, but rely purely on OS metrics.

One example for a feedback-based method is the implementation of the CLR (Common Language Runtime) generic thread pool for .NET4, which uses a combined hill-climbing and rule-based approach (called HC^3) to dynamically adjust the pool size in order to increase the work completions per second (throughput) [14]. While generally pursuing the same goal as our work, their work differs with regards to scope and approach. They aim to design a thread pool for generic jobs instead of only I/O-bound or CPU-bound work. This is in contrast to the earlier design of the .NET 3.5 concurrency controller, which assumed CPU-bound work. As a consequence of this change they do not use system resource metrics such as CPU utilization. With our focus on disk I/O-bound work only, it does make much more sense to base the approach on system resource metrics, as the definition of throughput can be narrowed to disk throughput. Regardless of these differences our approach is similar to the one by Hellerstein et al. as we also assume a concave structure for the concurrency-throughput function. Furthermore the controller states and non-collection of metrics during concurrency level changes are both inspired by their work.

More feedback-based work with a focus on thread pools in web servers and a target metric of latency has been done by Costa et al. [7]. They use roughly the same controller stages as the HC^3 controller, with decisions to scale up or down being based on observed latency of requests (which are tracked by another module). Chen at al. developed a linear search algorithm that uses indicators to decide which direction to scale to [3]. The algorithm consists of 2 phases, first the pool is scaled up until the

target performance metric stops increasing, then a potential downward adjustment is performed based on previously observed maximum and average latencies.

Finally the recent work by Khorasani et al. on adapting the currency levels of Spark executors for I/O-bound phases of data transformation was an inspiration to our work [18]. They show that the default settings of a thread count that equals the amount of CPU cores often performs much worse than optimal. They developed a feedback-based approach that dynamically adapts the concurrency level for I/O-bound phases which uses OS metrics for feedback on throughput and disk contention. Although their approach is much in line with ours, it is specific to the I/O access pattern used in Spark and uses a non-generalizable metric for determining contention (time spent blocking on the epoll_wait system call). Furthermore their controller algorithm assumes long job execution times, which are typical for the jobs executed by Spark executors, whereas our approach assumes jobs of a lower granularity.

Another set of prediction-based schemes that aim to adjust thread pool size according to the amount of requests using modelling techniques based on historical data try to minimize response times by predicting the future request rates. This approach was suggested by Kim et al. [19] and further developed by Kang et al. [15] and later Lee at al. [20]. These prediction-based schemes are all based on the assumption of optimal performance (response time in their case) when the amount of requests (submitted, not completed jobs) equals the amount of thread in the pool and they try to proactively create/destroy threads for the next predicted amount of requests.

Most of the above mentioned research relies exclusively on simulations or synthetic workloads to evaluate their implementations. While it is a good approach for iteratively developing algorithms, it may result in a discrepancy in performance when applied in real applications. We therefore integrate our solution in to two popular applications with workloads for specific use cases.

Another development complementary to the research on adaptive thread pools is the introduction of io_uring to Linux [1][5]. In an ongoing effort many blocking system calls are ported to it, enabling asynchronous programming with system calls that previously forced a synchronous programming style. This means asynchronous application architectures do not need a thread pool anymore for blocking disk I/O, they can be designed in a fully asynchronous manner. Recently there are many projects exploring this space, e.g the thread-per-core runtime Glommio, which relies on io_uring to provide a runtime for writing asynchronous code without the use of helper threads (that execute blocking code) [6]. While promising for future applications, it requires an application to be written in an asynchronous style, which may not be desirable due to complexity, or which may not be possible to achieve for existing applications.

4

Algorithmic Approach & Solution Architecture

In this chapter we introduce the algorithmic approach for an adaptive thread pool that minimizes runtime and amount of worker threads used. We first discuss how we picked appropriate OS metrics that we can use to implement a feedback-based adaptive thread pool. Based on the correlation of these metrics with the performance of the thread pool with different fixed sizes we introduce the algorithmic solution approach. We then show an overview of the design and components that make up the implementation, which is split into a generic thread pool and a scaling adapter that acts as the controller. To conclude the chapter we show experimental verification of the negligible overhead of the periodic execution of the adapter procedures.

4.1. System Metrics & Throughput Correlations

In this section we take a look at how the pool size of a fixed thread pool affects the total runtime of some deterministic workloads. Furthermore we investigate how the runtime over pool size function correlates with several system metrics over pool size. As we look at whole workloads we use the term throughput to mean the inverse of runtime, i.e the lower the runtime, the higher the throughput.

Two methods for collecting metrics are used, the `/proc` virtual filesystem is used to read some metrics that are exported by default by the Linux kernel, the systemtap tracer tool is used to observe which system calls the thread pool workers use, how often they are called and how much time is spent within the system calls in total [17][23]. All metrics are measured as aggregates over all the worker threads in the thread pool.

4.1.1. Workloads

For the purpose of investigation of throughput correlated system metrics and later for evaluation of our adaptive thread pool implementation we make use of several workloads. Every used workload in this thesis has a short description in the Appendix.

We wrote a few deterministic workloads that consist of homogeneous jobs, which perform disk I/O through the `write`/`read` system call interface. They differ in the size of files read and written, whether reads/writes are buffered, and how files are synced to disk (using the `fsync` system call). Our mutex-based thread pool with fixed size is used to execute the jobs, we run the same workload for pool sizes ranging from 1 to 64.

Furthermore we picked two popular open-source projects that make use of thread pools to execute disk I/O operations that can serve as good real-world examples. The first is NodeJS (from here onward just called Node), a Javascript runtime that makes use of a thread pool that is responsible for execution of all disk-related I/O work [29]. This thread pool and additional components are provided by the libuv project [26]. We implemented a few workloads that perform disk I/O by reading and writing files in bulk.

The second project is RocksDB, a key-value store that is heavily used as the core storage-engine by many other Database projects [10]. This project includes an extensive benchmarking suite with many workloads that capture a variety of usage patters [9]. RocksDB makes use of multiple thread pools, we

concentrate on the ”flush” pool, which is responsible for guaranteeing that internal database files are durably written to disk. In the following we show that for a write-heavy workload the size of this flush pool is having a significant impact on throughput.

4.1.2. Disk I/O Throughput

The Linux kernel reports per-thread I/O statistics through the `/proc/<tid>/io` files [17]. There are two different metrics for disk I/O throughput, `rchar` / `wchar` and `read_bytes` / `write_bytes`. `Rchar` and `wchar` are the amount of bytes a process has requested to be read/written from/to the disk-backed filesystem since its creation. `Read_bytes` and `write_bytes` are the amount of bytes that were actually caused to be read/written from/to the disk medium. These values may be different due to various factors such as the Linux page cache, read prefetch and other mechanisms.

The more appropriate pair to quantify disk I/O throughput from the application’s perspective are the `rchar` and `wchar` values. While there are workloads that mostly do writes and only read uncached files and therefore the two sets of metrics are not differing by much, more commonly these metrics would be different. For example a read-heavy workload may have most reads be served directly from the page cache, resulting in no actual disk reads, so the read_bytes metric does not capture application intent and actual performance, in contrast to the `rchar` metric. Thus, we focus on the `rchar`/`wchar` metric in the following, which we abbreviate as *rwchar* from here on. The `rchar` and `wchar` values are summed and aggregated over all worker threads and divided by the total runtime in seconds. The resulting value denotes the amount of read and written bytes per second for the whole thread pool, we call it the *rwchar* rate, which indicates the logical disk throughput. We define the *rwchar* rate (abbreviated R) over an interval $[t_1, t_2]$ to be:

$$R = \frac{(\sum_{w \in W}(rchar_{w,t_2} + wchar_{w,t_2}) - (rchar_{w,t_1} + wchar_{w,t_1}))}{(t_2 - t_1)}$$

where W is the set of workers in the pool at t_2 and $rchar_{w,t}$ and $wchar_{w,t}$ are the reported `rchar` and `wchar` values respectively for the worker thread w at time t (if the worker thread w does not exists yet at t, then $rchar_{w,t} = wchar_{w,t} = 0$).

***rwchar* rate - runtime/#threads correlation** Figure 4.1 shows both the runtime and *rwchar* rate over the amount of workers for 4 different workloads. The left y-axis shows the runtime in milliseconds, the right y-axis depicts the *rwchar* rate in bytes per second. The logical disk throughputs are highly correlated with the runtimes of the workloads, which is to be expected, since the these workloads purely perform either write or read operations to the disk. Runtime generally decreases with increasing amounts of worker threads up to a certain threshold where it stagnates and for some workloads runtime increases again with a high amount of workers. This is common behavior as noted in previous research [32]. The goal is to dynamically adapt the pool size to always be around the stagnation threshold, where a decrease in threads would decrease throughput, but a further increase would not improve throughput and put further load on the system.

As this optimal amount of threads is subject to change dynamically because of a change in the rate of job arrival, a change in job characteristic (e.g read vs write-heavy) or a change in system load due to external factors a thread pool should dynamically rescale according the the circumstances. Regardless of the algorithm used to optimize the pool size, the more stable and reliable the metrics used in the decision making, the better the performance.

***rwchar* behavior at runtime - increasing pool size** In order to gain insight into the behavior of the *rwchar* rate during execution we used a thread pool that increases in size with a steady rate of 1 extra worker thread per 2 seconds. The spawning of an extra thread is performed by the first worker that completes a job after the timeout has passed. The aggregated *rwchar* value over all workers is measured every 200ms. Instead of using the `/proc` interface to observe and aggregate the *rwchar* values we used a modified kernel with an added interface, a traceset, that can capture different metrics about a set of target threads. This interface is discussed in more detail later. While the *rwchar* metric is already exposed through `/proc`, there are more metrics we investigated that are not collected and exposed in the standard Linux kernel (in the next section we will have a look at these).

Figure 4.2 shows the *rwchar*/sec rate over time in red (left y-axis) and the pool size over time in blue (right y-axis). The green line shows the averaged *rwchar*/sec rate for the last 2000ms. We use the

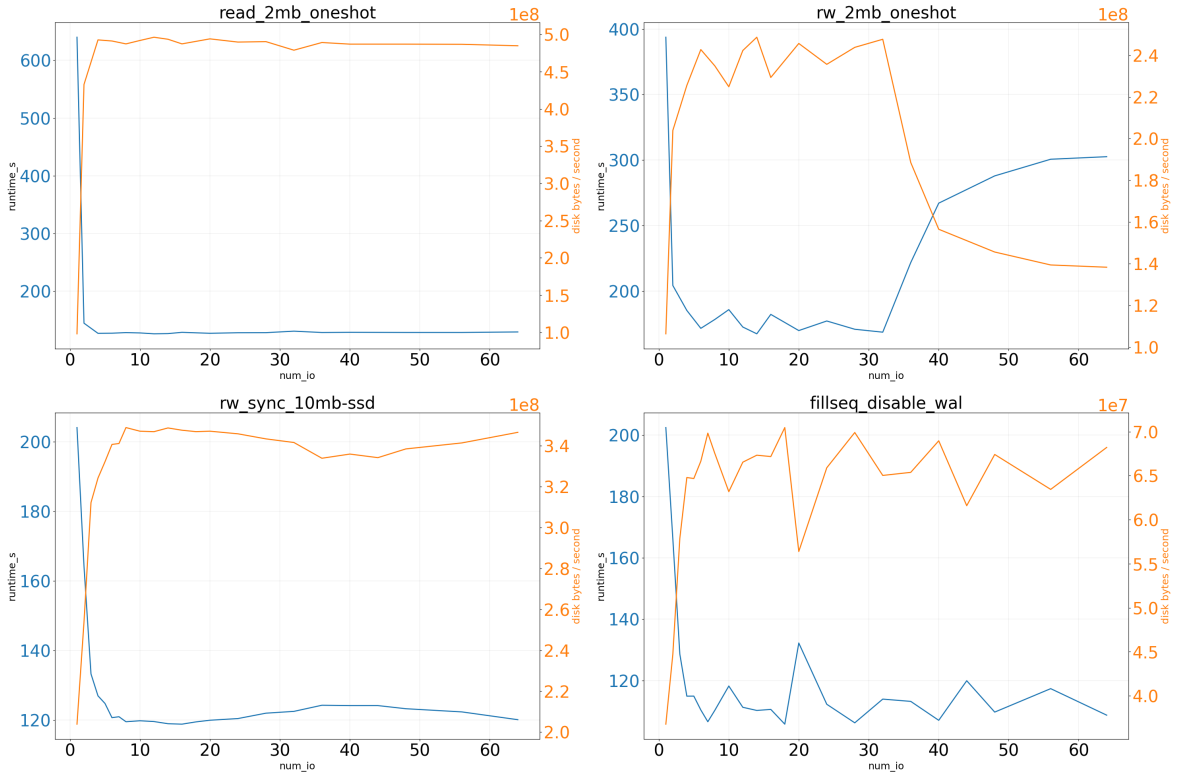


Figure 4.1: Runtimes and read/write throughput for various workloads

previously shown 2 workloads that read and write 2Mb files, the first workload's jobs just read single file into memory, the second workload's jobs read a single file into memory, then write the contents to a new file on disk and then synchronize it with the fsync system call. For the following sections we will refer to them as the read and the read-write workload. The workloads were run once and all jobs were submitted at the beginning, for the read workload X files are read, for the read-write workload it is X files.

Both workloads at first show an increase in *rwchar* rate as pool size is increased and reach a point of stagnation at some point. Furthermore, short peaks/drops of the *rwchar* rate that last a single interval can be observed for both workloads.

For the read workload logical disk throughput increases rapidly up to a pool size of 3 where it stagnates with small fluctuations and regular performance drops for a single interval. The averaged *rwchar* rate depicts a more stable view, with only small oscillations after a pool size of 3.

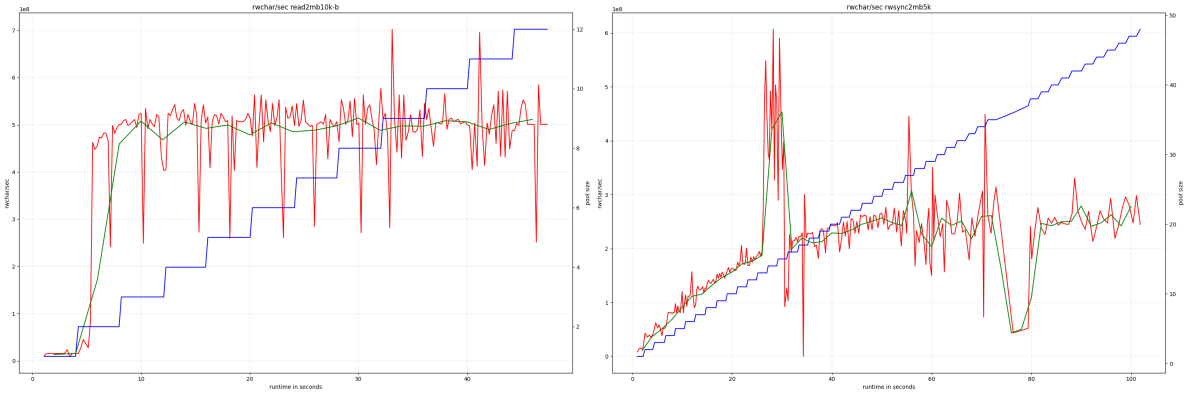


Figure 4.2: *rwchar*s/sec over time, pool size over time

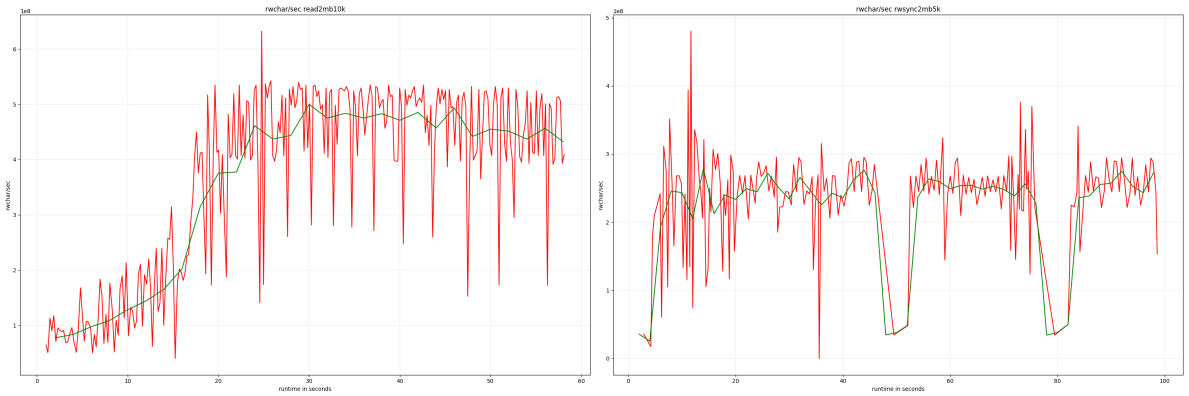


Figure 4.3: *rwchar*s/sec over time - fixed pool

For the read-write workload logical disk throughput rises slowly but steady up to a pool size of 15 where it suddenly doubles and peaks for about 6 seconds. Then at a pool size of 18 it drops again to a slightly higher level than before the peak and keeps slowly rising up to a pool size of about 25. The *rwchar* rate then stagnates and begins to fluctuate more with a prolonged drop of about 5 seconds starting at pool size 35.

Both experiments were repeated 3 times and while for the read workload behavior was consistent, the read-write workload showed prolonged peaks and drops in *rwchar* rate at different points in time for the different repetitions. However, the general trend of slow increase up to a point of stagnation and then an increase in variability was the same.

***rwchar* behavior at runtime - fixed pool size** We did a second experiment with a fixed-size thread pool to confirm that the prolonged intervals of performance drops are inherent in the read-write workload and not directly triggered by the previous increase in pool size. Figure 4.3 shows the same workloads and their *rwchar* rate over time.

Surprisingly, the read workload's *rwchar* rate does increase in the first 30 seconds until it finally hits the peak at 500Mb/sec equivalent to the peak rate of the previous experiment. We are unsure why the rate is not constant around peak performance right from the beginning.

For the read-write workload the *rwchar* rate is peaking almost right from the start and two prolonged drops occur at around 45 and 75 seconds, just like for the previous experiment. It seems that at those moments the disk performance suddenly deteriorates significantly and then recovers after a while. This behavior complicates any approach that is based on maximizing the *rwchar* rate in order to achieve an optimal pool size, so we also investigated alternative metrics.

4.1.3. Auxiliary Metrics

Apart from disk throughput we also investigated additional metrics that show a correlation with the thread count over runtime function. For the mentioned workloads in the previous section the major system calls associated with disk operations are the `read`, `write` and `fsync` system calls. We call these the I/O syscalls, and the amount of I/O syscalls issued per second the I/O syscall rate. The I/O syscall rate exhibits a similar correlation to total runtime w.r.t to the amount of threads in the thread pool. Figure 4.4 shows the I/O syscalls rate and total runtime w.r.t to the pool size for the same 4 workloads from the previous section. Just like the *rwchar* throughput the I/O syscalls rate over pool size function is negatively correlated with the runtime over pool size function.

Just as in the previous section we use a steadily increasing thread pool to investigate the behavior of the I/O syscall rate during execution of two read-write workloads. Figure 4.5 depicts the I/O syscall rate over time and figure 4.6 shows the average execution time of I/O system calls over time.

The workloads are the same as in the previous section, figure 4.5 shows the amount of I/O syscalls per second on the y-axis in red, figure 4.6 shows the average time in milliseconds spent per I/O syscall in the respective interval on the y-axis in red. For both workloads the behavior of the I/O system call rate is almost identical to the *rwchar* rate with no significant difference.

As seen earlier the read workload's I/O system call rate exhibits more stable behavior as the pool

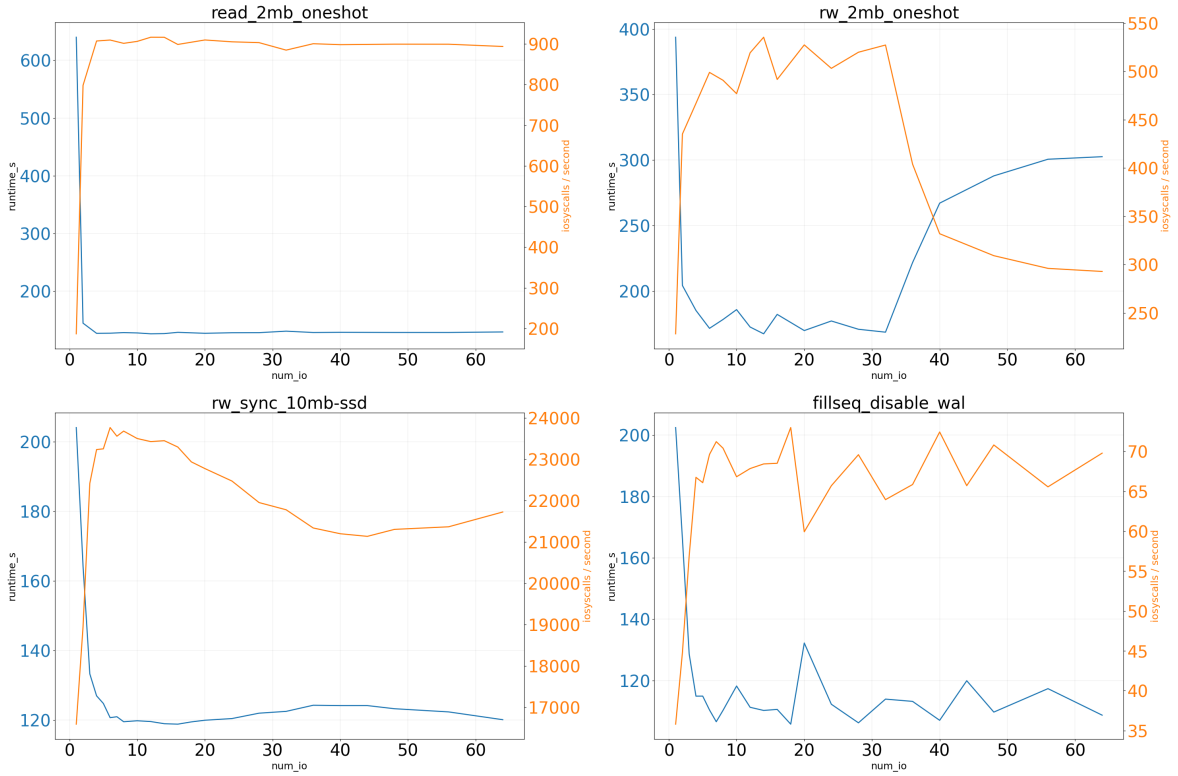


Figure 4.4: Runtimes and iosyscalls rate for various workloads

size increases. The average call time sharply drops to a minimum after the pool size is increased to 2 and then slowly rises with increasing pool size. Small peaks at the same times when the *rwchar* rate drops shortly are observed. For the read-write workload's period of dropped logical disk throughput the amount of I/O syscalls per second also sharply drops and the average execution time conversely peaks at around 7 times the previous value.

Overall the I/O system call rate does not offer any advantage over the *rwchar* as the target metric to maximize. Other metrics and combinations of metrics were investigated without any showing a similar correlation to total runtime as well as being more stable during workload execution. Therefore our solution is based solely on maximization of the *rwchar* rate.

4.2. Algorithmic Approach

As we have shown in the previous section, the *rwchar* rate is highly correlated with general throughput for IO-heavy workloads. Therefore we base the algorithmic approach on the general goal of maximizing *rwchar* rate. The thread pool controller can continuously monitor its value for short intervals and dynamically adjust the pool size to maximize it. This is done in a hillclimbing fashion, when the adapter decides to start a scaling phase, the pool size is continuously adjusted upwards or downwards until no better throughput is observed. As the *rwchar* rate is prone to fluctuation, especially when the pool size changes, we selectively filter out the intervals around the pool size adaption times. Additional phases besides the scaling phase are introduced to continue to react to the observed throughput.

In the following we discuss some assumptions we base the algorithm on and the resulting limitations. Then we introduce the general approach and the detailed implementation.

4.2.1. Assumptions and Limitations

Our method is based on some basic assumptions and is limited to certain use cases, which we shortly discuss here before discussing our approach.

Our approach is based on hillclimbing and feedback through OS metrics, which requires the workload to consist of relatively stable periods of similar work to be performed by the thread pool, adapting to

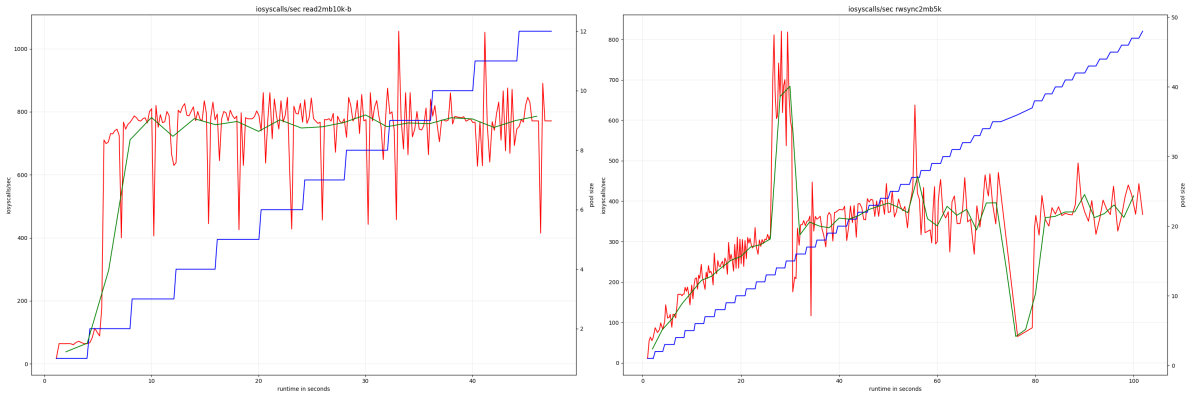


Figure 4.5: I/O syscall rate and pool size over time

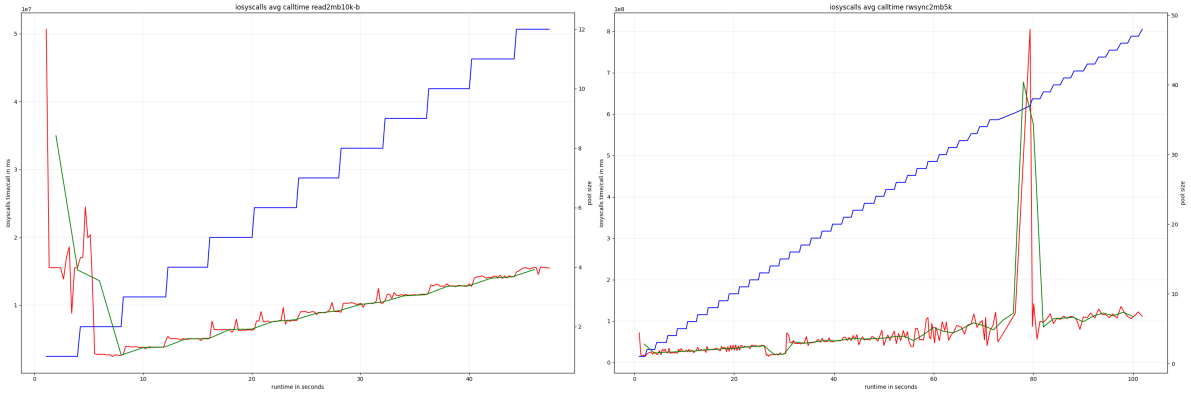


Figure 4.6: I/O syscall avg time/call and pool size over time

new circumstances takes at least several seconds. So our approach is effectively limited to these kind of scenarios and will underperform in highly variable and fast changing workloads. For our experimental analysis we thus limit ourselves to this use case.

Furthermore we assume that for a given deterministic workload (a precise definition follows in the next chapter) the runtime/pool size function is approximately convex downward, whereas the function disk throughput/pool size is approximately concave upward. This assumption roughly holds from the perspective of total runtime for different thread pool sizes as shown in figure 4.1. When sampling the throughput during runtime, just like the controller would do, the throughput curve is clearly not convex due to intervals of fluctuation. However, we found that in practice our solution still performed well even though the assumption is violated in many cases.

4.2.2. Adapter Algorithm

The controller switches between different states depending on the *rwchar* rate it observes, attempting to maximize the throughput while keeping the pool size low. Generally it will be either in a phase of scaling the thread pool down/up, settled on a fixed size or exploring smaller/bigger pool sizes. The controller state is reevaluated at fixed intervals, according to the *rwchar* rate over the respective last interval. Figure 4.7 illustrates the whole state machine that defines states and state changes of the controller.

When first initialized, the controller is in *Startup* state, where it increases the pool size by 1 and then changes its state to the *Scaling* state with step size 2. The thread pool should start with a size of 1, which is thus unconditionally increased to 2. At startup it is necessary to unconditionally scale up the pool in order to have two different intervals with different pool sizes to compare to each other. An alternative strategy may be to start with a high thread count and adjust downwards, but it introduces another configuration parameter and was therefore not further explored.

During the *Scaling* phase the controller compares the average *rwchar* rate over a fixed interval to

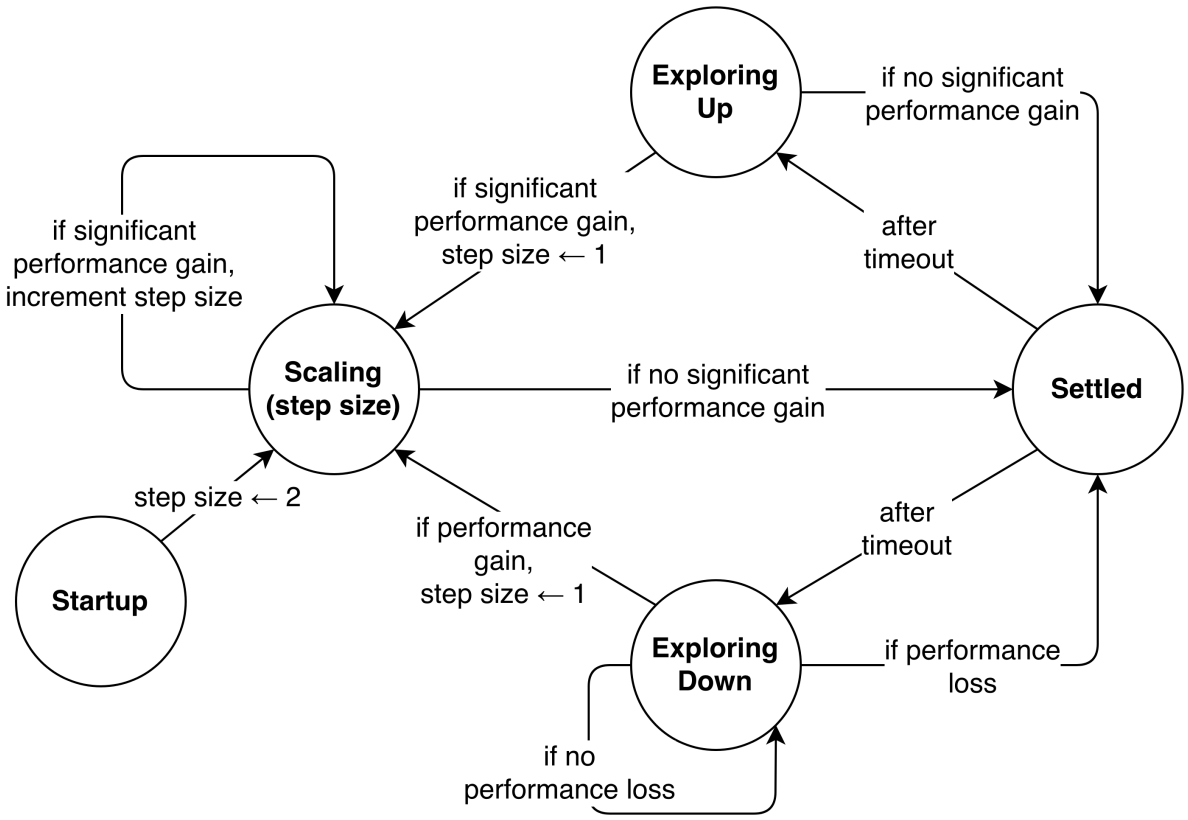


Figure 4.7: States of the controller

the previous interval and keeps increasing the thread pool size until no significant performance gain is observed anymore. When in the *Scaling* phase the controller continues to adjust to thread pool size upwards or downwards depending on the scale direction. In order to find the optimal pool size more quickly, the amount of threads added/removed (step_size) increases at every additional *Scaling* phase (up to some limit). When no significant performance gain is observed, the controller switches to the *Settled* state.

Once *Settled*, after a timeout, the controller explores the opposite direction w.r.t the last *Scaling* phase or the last exploration move, depending on which of the two is more recent. When in the *Exploring* state the controller adjusts the thread pool size by 1 down or up.

For both explore directions the controller switches to the *Scaling* state in case of significant performance gain in order to more quickly adjust towards that direction. If exploring downwards and no performance loss is observed, the controller will continue to explore downwards, reducing the pool size by 1 and remaining in the same state. The reasoning here is that while performance remains stable the pool size should be minimized as much as possible, without risking too much performance loss by reducing the pool size too much.

The factor (stability factor) determining what difference is significant is a core parameter to the algorithm and was chosen after experimentation for a variety of different workloads. The other important parameter is the interval length, which determines the time frame over which the *rwchar* rate is determined and also the frequency of potential adjustments to the thread pool size. Below we list the parameters that were chosen experimentally by testing different combinations for most of the workloads mentioned in this thesis.

- stability factor (tested values ranging from 0.85 to 0.97)
- interval length (tested values from 500ms to 3000ms)
- starting step size, step sized increase, step size limit: (tested values from 1 to 2, 1 to 2 and 2 to 8)

We further describe the process of parameter selection in section 5.2.2.

4.2.3. Implementation

The controller algorithm and state are implemented as part of a stateful adapter module, which is discussed in the following section. The behavior is encapsulated in a procedure that needs to be called periodically by any of the worker threads in the thread pool and returns the amount of threads that should be added or removed. Algorithm 1 shows pseudocode for this procedure. Metrics are recorded over smaller intervals (subintervals) to facilitate filtering of the first short interval after pool size has changed. The metrics are then averaged over the configurable scale interval in the routine that updates the interval history. If the configured scale interval has passed, the specific subroutine corresponding to the current state is called, otherwise the adapter advises no change to the workers.

Algorithm 2, 3 and 4 show the procedures that implement the control mechanism for each of the *Scaling*, *Settled* and *Exploring* state.

```
   Output: Amount of threads to add/remove
 1 if subinterval passed then
 2 |   update subinterval history
 3 end
 4 else
 5 |   return 0
 6 end
 7 if interval passed then
 8 |   update interval history
 9 |   switch state do
10 |   |   case Startup do
11 |   |   |   state ← Scaling(2)
12 |   |   |   return 1
13 |   |   end
14 |   |   case Scaling(step_size) do
15 |   |   |   return getAdviceScaling(step_size)
16 |   |   end
17 |   |   case Settled(timeout, direction) do
18 |   |   |   if timeout has passed then
19 |   |   |   |   return getAdviceSettled(direction)
20 |   |   |   end
21 |   |   |   else
22 |   |   |   |   return 0
23 |   |   |   end
24 |   |   end
25 |   |   case Exploring(direction) do
26 |   |   |   return getAdviceExploring(direction)
27 |   |   end
28 |   end
29 end
30 else
31 |   return 0
32 end
```

Algorithm 1: get scaling advice

4.3. Architecture & Components

In this section we briefly discuss the concrete implementation of the scaling adapter and its integration into a thread pool to make it adaptive. The implementation of the adaptive I/O thread pool consists of multiple components. The logic that determines whether the pool should adjust its size is contained in

```
   Input: step_size
   Output: Amount of threads to add/remove
 1 if perf significantly higher then
 2 |   state ← Scaling (step_size + 1)
 3 |   return step_size + 1
 4 else if perf significantly lower then
 5 |   state ← Settled(direction)
 6 |   return step_size
 7 else
 8 |   state ← Settled(direction)
 9 |   return 0
10 end
```

Algorithm 2: get scaling advice - Scaling

```
   Input: Last direction explored/scaled: last_direction
   Output: Amount of threads to add/remove
 1 if perf significantly higher then
 2 |   new_direction ← UP
 3 else if perf significantly lower OR significantly more fluctuating then
 4 |   new_direction ← DOWN
 5 else
 6 |   new_direction ← opposite of last_direction
 7 end
 8 if new_direction is UP then
 9 |   state ← EXPLORING_UP
10 |   return 1
11 end
12 else
13 |   state ← EXPLORING_DOWN
14 |   return −1
15 end
```

Algorithm 3: get scaling advice - Settled

```
   Input: Direction explored: direction
   Output: Amount of threads to add/remove
 1 if direction is UP then
 2 |   step_size ← 1
 3 end
 4 else
 5 |   step_size ← −1
 6 end
 7 if perf significantly higher then
 8 |   state ← Scaling(step_size)
 9 |   return step_size
10 else if direction is UP OR (direction is DOWN AND perf lower) then
11 |   timeout ← getTimeout ()
12 |   state ← Settled(timeout, direction)
13 |   return -step_size
14 else
15 |   return step_size
16 end
```

Algorithm 4: get scaling advice - Exploring

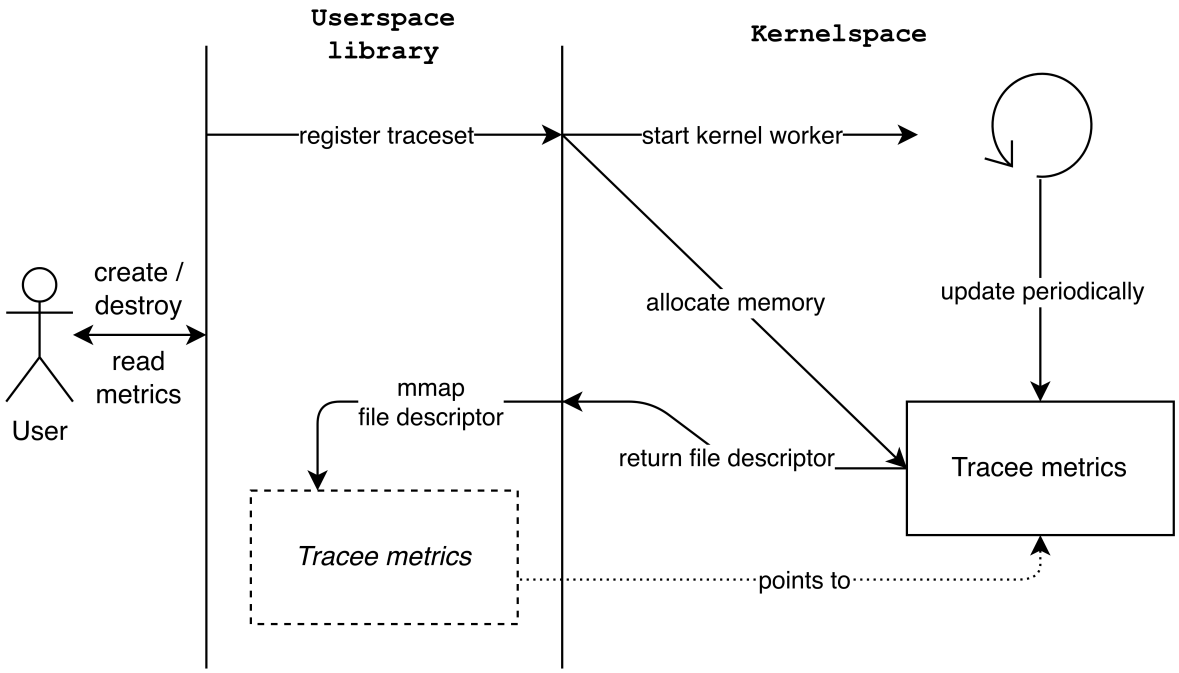


Figure 4.8: Architecture of Traceset

an adapter module which can be added with little friction to an existing static thread pool. It isolates the control algorithm from the specifics of the thread pool implementation, so it is agnostic of the synchronization mechanism and job queue details. The adapter exposes methods to register workers to be traced and a single method to obtain advice on whether to scale the pool size.

System metrics are obtained through the use of a traceset, which defines a mutable set of threads whose metrics are collected and their respective values, which are being updated continuously. The scaling adapter adjusts the set of traced threads and uses the metrics for the control algorithm. The traceset functionality resides mostly in the kernel, with a small user space wrapper library.

We implemented two thread pools with different synchronization mechanisms, one based on spinlocks, one based on mutexes. Furthermore we integrated the scale adapter into two existing applications that use thread pools for purely disk I/O tasks.

4.3.1. Tracesets

In order to efficiently collect information regarding disk usage and system call information on a per-thread basis, a small library consisting of some kernel modifications including the addition of 2 system calls and some user space wrappers was developed. While the *rwchar* metric could be collected by using the `/proc` interface, other metrics such as system call information have to be actively traced. Figure 4.8 shows a diagram illustrating the interface and implementation of the traceset functionality.

Kernel Space To provide the user program the means to register/deregister threads to be traced two system calls `traceset_register` and `traceset_deregister` were added to the Linux kernel. This interface enables a process to start/end tracing of any of its child threads and provides access to the gathered statistics trough shared memory. The conditional tracing is achieved through additional fields in the `task_struct` for saving the information and checks whether a process and system call is traced in the generic system call handler that dispatches requests to the respective system call handler. The data regarding system call usage and disk usage present in each `task_struct` serves as the source for the shared memory that is accessible from user space. As long as there are some processes being traced a recurring update handler is run to keep them in sync. It is implemented with the event workqueue and a single updater function that reschedules itself.

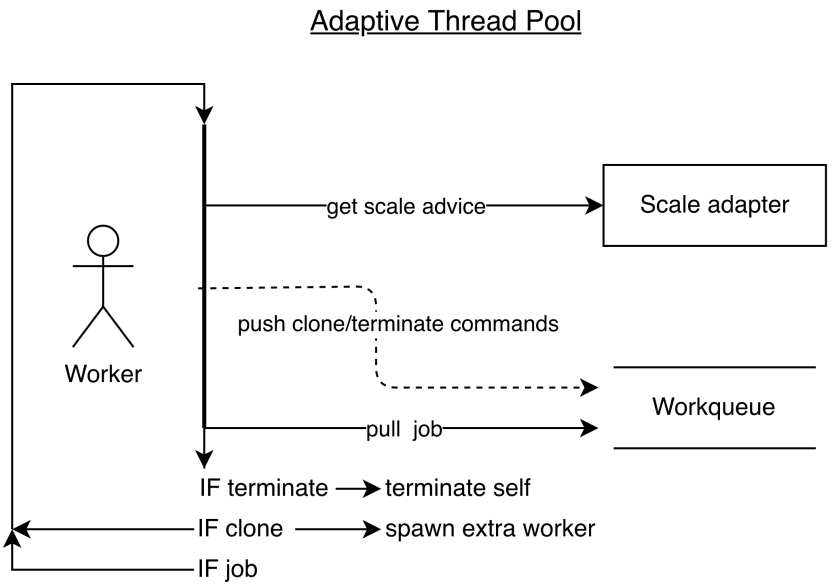


Figure 4.9: Architecture of Thread pool

User Space To simplify usage of tracesets in user programs the system calls are wrapped with more user-friendly functions which are written in C. On top of these an object-oriented interface is implemented in Rust.

4.3.2. Scale Adapter

The scale adapter is the core component that encapsulates the behavior and state needed to make the decision how to adjust the thread pool size in order to increase throughput. It saves the history of relevant OS metrics, manages the traceset associated with the worker threads in the thread pool and provides the interface to obtain advice on how to scale the pool. A thread pool that integrates the scale adapter and bases pool size adaption actions on it must periodically call the scale adapter to let it update its internal state and return advice on the amount of threads that should be added or removed. Furthermore, the pool's worker threads must register with the adapter as a traceset target on startup and deregister on termination.

4.3.3. Thread Pool

We implemented two versions of a basic thread pool leveraging the scale adapter to automatically adjust the amount of worker threads to the current workload. The two versions differ solely in the use of synchronization mechanism, one version uses mutexes and condition variables, whereas the other one uses spinlocks. As most implementations are mutex-based, including the ones from the example applications investigated, for all experimental analysis the mutex-based version is used.

Work items are user-supplied functions or scaling commands. The items are queued in a synchronized queue, which is consumed by the worker threads. Workers execute a loop, continuously pulling new work from the queue and executing it. Every iteration they query the scaling adapter for potential readjustment of the pool size. If the scaling adapter advises to scale up or down, the worker that received the advice pushes the appropriate amount of clone/terminate commands to the front of the work queue. The loop and the different components are illustrated in figure 4.9.

This design imposes a restriction on the nature of the jobs that are executed, they should not take much longer than the chosen adapter advice interval. If the execution of the jobs takes longer, the scaling advice procedure may only be called after much longer intervals as all workers are busy executing a job. Furthermore, when in a phase of scaling down with a large step size it may take a long time from the moment that the scale down advice is given until the requested amount of workers has terminated. This can only possibly be avoided if jobs are allowed to be interrupted and continued by different workers.

For one of our example applications, RocksDB, in the workloads we have investigated the flush

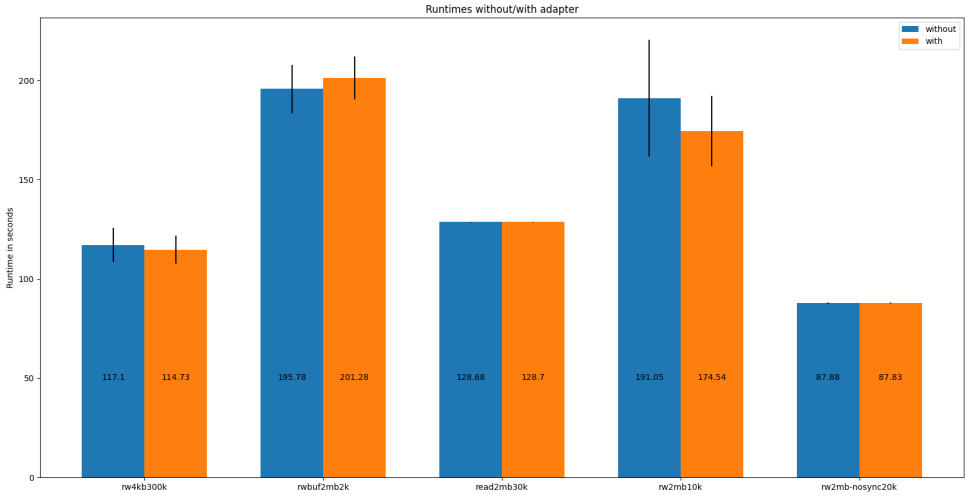


Figure 4.10: Overhead of adapter

pool does execute long running jobs, which forced us to adopt a different design when integrating the adapter into the pool. An extra thread is used to periodically trigger the scaling advice procedure and to spawn new threads and instruct workers to terminate. We will discuss the two different implementations in more detail in section 5.3.4.

4.4. Overhead of Scaling Adapter

One issue with feedback-based approaches is the potential overhead that the controller may introduce. In order for our implementation to be effective, regardless of how accurate the adapter is in controlling the pool size to be close to optimal, the addition of the adapter should introduce very little overhead. Specifically the execution time of the scale advice method should be kept as short as possible. Additionally the recurring execution of the kernel thread that updates the traceset data should be take up little processing time to avoid influencing the amount of processing time given to the thread pool workers.

In the following we compare the runtime of the normal fixed size thread pool with a modified version that calls the scale advice method in the same fashion as the adaptive pool. The extra work performed by this modified version corresponds almost exactly to the extra work of the controller activities in the adaptive pool, only the insertion and parsing of scaling commands are missing.

We use 5 different workloads of reading/writing files of different sizes and with different ways of performing the reads/write (buffered/non-buffered and synced/not synced). For all workloads the whole set of jobs is submitted upfront to the thread pool and they terminate once all jobs have been processed. The thread pool size is fixed to 16 for 2 of the workloads and 32 for the others. Each experiment was repeated for 10 times.

Figure 4.10 shows the results for all the workloads, the y-axis depicts the total runtime for the different workloads on the x-axis. Barplots are used to show the average runtime over the 10 runs, the error-bars depict the standard deviation. The fixed pool without the adapter is visualized with the blue bars, the fixed pool with the adapter is orange.

For none of the workloads there is a significant difference in runtime, whether using the scale adapter or not. The workloads with higher standard deviation show slight differences, which are so small (in contrast to the standard deviations) to be statistically insignificant. The two workloads with almost no variance (read-only and read-write without sync) do not show any difference between the two thread pool versions.

It can be concluded that with the proposed implementation of the scaling adapter the goal of no significant overhead is achieved.

5

Workloads & Experimental Analysis

In this chapter we first define a descriptive model for workloads that captures workloads with multiple phases and characterizes phases and phase changes. This model is used to characterize the different workloads used for experimental evaluation of the adaptive pool performances, as well as to define potential upper bounds on performance. We then introduce the experimental setup and shortly discuss our methodology for the analysis and picking of adapter parameters.

The experimental analysis is split into different sections for synthetic workloads that were executed on our own adaptive pool implementation, and the workloads for the modified Node / RocksDB applications. We distinguish between single-phase and multi-phase workloads as defined in the first section and analyze total runtimes, average pool size and the scaling behavior of the adaptive pool implementations. For the workloads that test our adaptive pool implementation we compare the runtime and pool size with the optimal fixed-size pool and the Watermark model pool, for the application workloads we compare with the default fixed-size pool and the optimal fixed-size pool.

5.1. Modelling Workloads

In the following we introduce a simple model of workloads consist of one or more intervals (phases) where each interval is stable w.r.t to load and I/O access patterns. First and foremost a workload is defined by an ordered list of job and submission time tuples, where jobs are ordered by increasing job submission times. A job is just a tuple of a procedure without return value and its input arguments. When modelling and describing workloads in the following sections we simplify the definition of a job to be given solely by the procedure run (job type). A workload consists of one or more consecutive phases, where each phase has a start and end time corresponding to the submission times of two jobs.

5.1.1. Phases & Phase Changes

We characterize a phase by the following 3 characteristics that are fixed during the whole duration of the phase:

- Job submission rate (amount of jobs submitted per second)
- Job type
- Background system load

With this restricted model we aim to capture the main scenarios where an adaptive thread pool should adjust its pool size in order to adapt to changed demands. For each phase an optimal pool size exists as the environment is stable, and for each new phase this optimal size may change. We can experimentally determine the optimal pool sizes and check if the adaptive pool correctly identifies these and how quick this happens.

This phase-based model for workloads is a simplification to aid the analysis of the adaptive thread pool, but we also think that in order to achieve acceptable performance that reliably outperforms default fixed-size configurations, workloads have to roughly conform to this model of relatively stable phases

that last at least a few seconds. Especially for batch data transformation workloads that persist data to disk in between the different transformation phases, we believe the phase-based model to be a suitable characterization. With the synthetic read-write workloads used in the experimental analysis we aim to model such data transformation workloads in a simplified manner.

For more unstable and highly variable workloads our feedback-based approach may not be very effective. Generally we are not convinced a generalizable adaptive scheme for such workloads can perform well if no prior knowledge about the characteristics (job types / submission rates) are known.

5.1.2. Saturating vs Non-saturating Phases

We distinguish between saturating and non-saturating phases, in a saturating phase the queue size of the thread pool is constantly rising as an upper limit of the rate of processed jobs is reached. In a non-saturating phase the queue size is either shrinking or staying around 0 constantly, the disk is not necessarily fully utilized and the optimal amount of worker threads is equal to the amount that is needed to process all jobs in time.

For the non-saturating case our adaptive pool should perform similarly to the Watermark model and offer no advantage over it. Therefore we mostly concentrate on workloads with saturating phases in the following analysis. We also show on example of a multi-phase workload with a non-saturating phase, where the adaptive pool performs similar to the Watermark model.

5.2. Methodology & Experimental Setup

In our experimental analysis we analyze various workloads that were developed to experimentally guide and evaluate the scaling adapter implementation. We use the average runtime for a single workload as a performance grade. The adaptive pool implementation is compared against the non-adaptive fixed-size version of a thread pool with exactly the same architecture (sharing much of the code). For stable load workloads with homogeneous jobs the runtime of the best performing static size thread pool version serves as a theoretical lower-bound for the same workload using an adaptive thread pool.

In order to evaluate the performance of our solution approach, we experimentally test the scale adapter in 3 different contexts. We use file read/write workloads for our adaptive thread pool implementation and for Node with the scale adapter integrated into the Libuv thread pool, and write-heavy insertion/update workloads for RocksDB with a modified flush thread pool.

For our own adaptive thread pool implementation we also implemented a version that uses the Watermark adaption scheme. So for all the workloads tested on our thread pool implementation we contrast our adaptive scheme with the Watermark scheme. Furthermore, for all analyzed workloads over the 3 different applications we experimentally determined the optimal pool size when using a fixed-size pool. Here optimal means lowest pool size while having a total runtime of maximum 3 percent higher than the fastest pool size configuration. For single-phase workloads the fixed-size pool with optimal configuration is expected to execute the workload the fastest, therefore serving as an upper limit in terms of throughput.

5.2.1. Experimental Setup

Total runtime and average pool size over the whole execution are investigated, as these constitute the two target metrics we optimize for. To determine runtimes every workload is executed 3 times and mean and standard deviation are shown in the graphs. Average pool size is inferred for fixed-size and Watermark pool, for our adaptive pool an execution trace is used to compute it.

All experiments were run on a machine with an Intel i5-6500 CPU with 4 cores at 3.2GHz clock rate and a 6MB cache, 16GB DDR4 memory and an SSD.

5.2.2. Selection of Algorithm Parameters

To iteratively evaluate our adapter implementation during development we built a set of scripts that runs new versions / new parameter combinations over the whole set of developed benchmarks. In order to pick the best parameter values for the adapter algorithm different combinations of stability factor and interval length were tested with this setup, stability factors ranging from 0.85 to 0.98 and interval lengths ranging from 500ms to 3000ms were tried. For each adaptive pool implementation the parameter combination that resulted in lowest runtime for the respective workloads was picked. Other constant factors used in the algorithm were not extensively tested, but chosen during development

according to just a few test runs. In the beginning of the sections for our own adaptive pool, Node and RocksDB we state the specific values of these parameters.

Since the best parameter values differ for different workloads we think one approach to improving average performance over a wide range of workloads would be to dynamically choose and adapt these parameters. Particularly the interval length may be chosen according to current average execution time of single jobs. Further research on this is deferred to future work.

5.3. Experimental Analysis

In the following the analyze the performance of our adaptive thread pool implementation and the scaling adapter integrations into the Libuv thread pool used in Node and the RocksDB flush thread pool. We begin the discussion of the experimental results with our own thread pool implementation before continuing to the modified applications.

5.3.1. Adaptive Thread Pool - Single Phase

We first take a look at 3 workloads that read/write files of 2Mb size from/to disk. All workloads consist of a single phase of homogeneous jobs that are submitted all in one batch upfront, so they are oversaturating. Two workloads were previously used to illustrate the correlation of runtime with OS metrics, figure 4.1 shows the performance of the fixed size pool for different pool sizes.

1. read_2mb: read a 2mb file from disk
2. rw_sync_2mb: read a 2mb file from disk fully, then write content to new file fully before calling fsync on it
3. rw_buf_sync_2mb: read 4kb from 2mb file into buffer, then write 4kb buffer to second file and call `fsync`, repeat until whole file has been written

For all these workloads and the multi-phase workloads in the following section the adaptive pool is instantiated with a 800ms scale interval and a stability factor of 0.97.

Figure 5.1 shows a comparison of fixed pool (blue), Watermark pool (orange) and adaptive pool (green). The Watermark pool was configured with a maximum size of 64 threads. The first graph depicts the mean runtimes for the 3 workloads, standard deviations are shown with the black error bars.

For the first workload, rw_sync_2mb, we can see that the adaptive pool performs almost identical to the optimal fixed-size pool, with statistically insignificant lower runtime and more variance. The Watermark pool's runtime is almost twice as high, which confirms the trend of sharp increase in runtime that was already shown in previous chapter's figure 4.1. The buffered version of this workload, rw_buf_sync_2mb, shows different behavior, the Watermark pool performs just marginally worse than the optimal fixed-pool, whereas the adaptive pool's average execution time is about 25% higher than the optimal fixed-pool. The picture is similar for the last workload, read_2mb, however, the adaptive pool performs only about 10% slower than the optimal fixed pool here.

The second graph compares the average pool size over the whole execution for the fixed pool (blue), Watermark pool (orange) and adaptive pool (green). As the workloads are all saturating, the Watermark pool immediately scales up to the maximum amount of threads and continues to use all workers to execute jobs. Here we see that for the first and last workload, as the adaptive pool manages to scale the pool to near optimal size, it also performs as well or relatively close to the best possible. For the buffered read-write workload the adaptive pool uses too few worker threads to approach optimal performance.

Overall, these workloads confirm that the Watermark model is too simplistic for the purpose of disk I/O only job execution. When the optimal pool size is overshot, in the worst case throughput takes a large hit, in the best case still a lot of memory is wasted through excessive use of threads.

Finally we take a look at the scaling behavior of the adaptive pool during the execution of the 2 read-write workloads. Figure 5.2 shows both the pool size (in blue) and the *rwchar* rate (in red) over the whole execution of the workloads. The buffered read-write workload is shown on the top, the unbuffered one on the bottom. The reported value of the *rwchar* rate r_i at a certain point in time t_i describes the rate over the interval since the point in time t_{i-1} of the last reported value r_{i-1}.

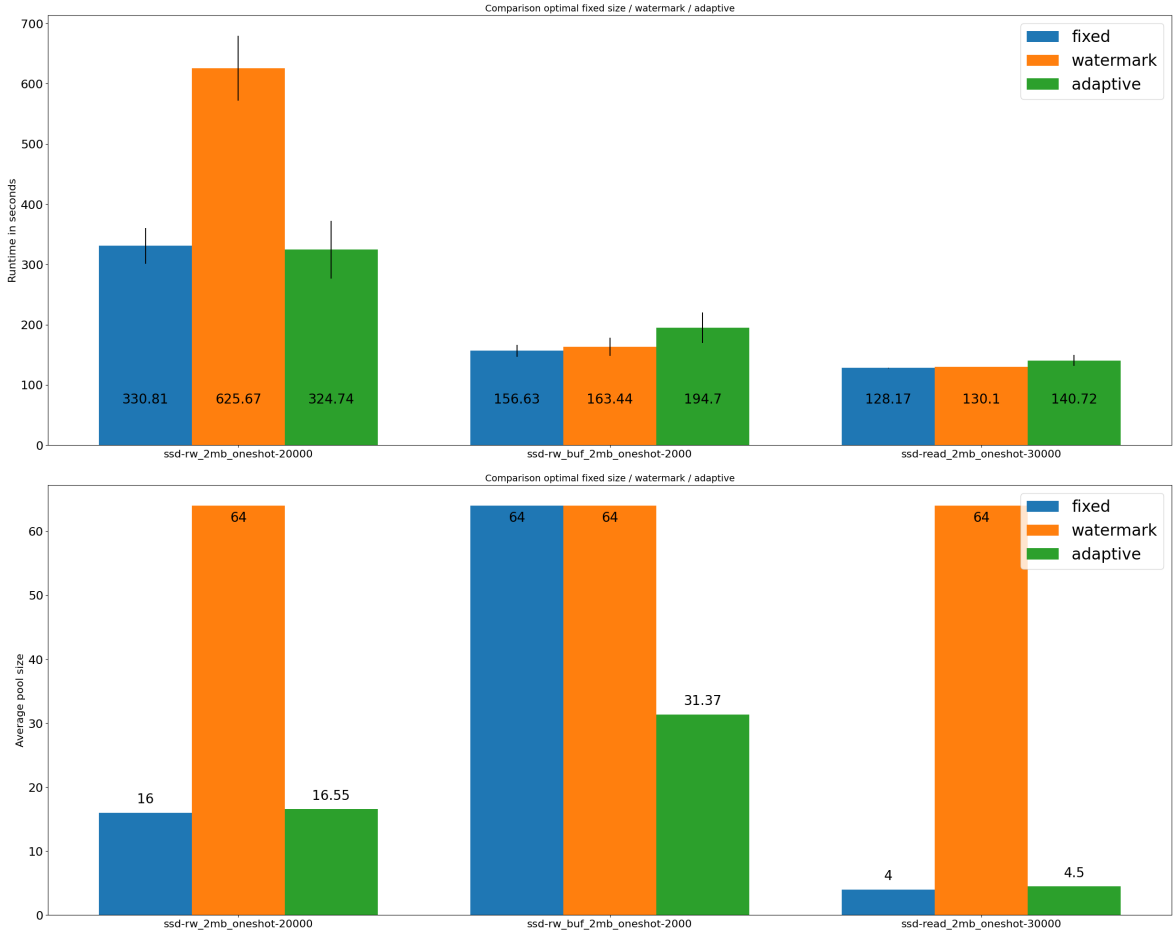


Figure 5.1: Comparison of optimal fixed size, Watermark, adaptive pool

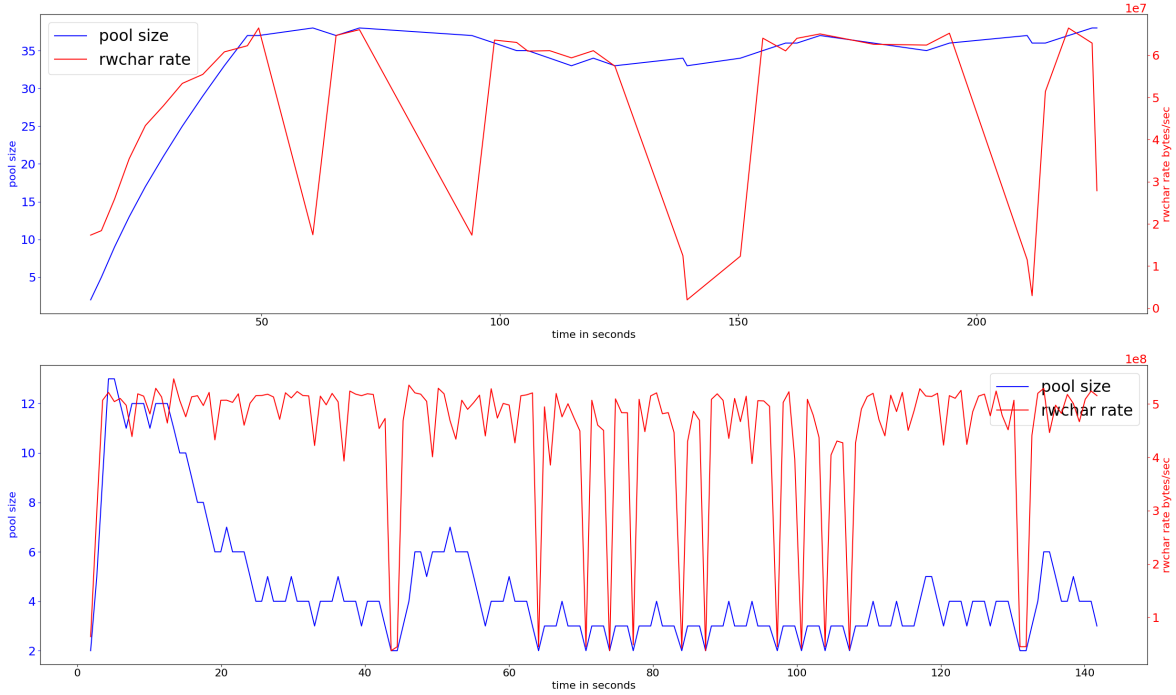


Figure 5.2: Adaptive thread pool runtime behavior

Buffered read-write Within the first 50 seconds the *rwchar* rate steadily increases and the adapter continuously scales up the thread pool up to a size of around 36 threads. The next interval is much longer, all worker threads are busy executing their jobs and no new scale action is triggered for more than 10 seconds, once the next scale action is triggered the *rwchar* rate dropped substantially, resulting in an explore downwards move. This pattern of *rwchar* rate drops that cause long intervals without the scale actions repeats for the remainder of the execution, interleaved with stable peak performance and shorter intervals. In general however, even for the stable periods, the minimum interval length is much longer than the actual desired interval of 800ms, due to the jobs taking a longer amount to execute. As we noted before, this is a fundamental restriction with the architecture of the worker threads triggering scale actions themselves.

Unbuffered read-write The unbuffered read-write jobs are executed within the desired 800ms intervals, so the thread pool is scaled up quickly to 13 threads within the first 5 seconds. Due to the *rwchar* rate not decreasing on exploring downwards moves, the pool is scaled back to a size of 4 and later 3, staying settled around these sizes. After around 45 seconds the first major drop in *rwchar* occurs for the length of 2 intervals, the pool is scaled down and then scaled back up as *rwchar* rate recovers. This pattern repeats over much of the remainder of the execution.

From the behavior of the adaptive pool throughout these 2 workloads we conclude there are 2 significant problems that negatively impact the scaling behavior. The first one is jobs with longer execution time, which is just a problem for some workloads, the second one are occasional large drops in the target metric.

5.3.2. Adaptive Thread Pool - Multi Phase

For single-phase workloads the adaptive thread pool should theoretically not outperform the best fixed-size pool, but for multi-phase workloads this may be different. If the optimal pool size differs for the different phases, with an adaptive approach the optimal pool sizes for each phase can possibly be reached by rescaling, whereas the fixed size pool will perform suboptimal for some or all phases.

Unbuffered-buffered We combine the two read-write workloads from the previous section to a multi-phase workload, where the phases are distinguished by the different job types. This workload has 3 phases, phase 1 and 3 consist of unbuffered read-write jobs, during phase 2 the read-write jobs are buffered, we call it rw_rwbuf_rw-2mb.

The left graph in figure 5.3 shows a comparison of total runtime for the different optimal sized fixed pools (size 16: optimal for unbuffered read-write, size 64: optimal for buffered read-write, size 32: optimal for combined multi-phase workload) with the adaptive thread pool. The adaptive pool performs better than the fixed-size pool with optimal sizes for phase 1 & 3 and phase 2 respectively, but it outperformed by the optimal (w.r.t to the whole workload) fixed-size pool. The right graph of figure 5.3 shows the pool size and *rwchar* rate over time for the adaptive pool executing the rw_rwbuf_rw-2mb workload. The 3 phases are evident from the much lower average *rwchar* rate from around 120-270 seconds, which is marks phase 2 where the buffered read-write jobs are executed. For that phase the pool size ranges from 25 to 30, the scale adapter fails to scale the pool size to the optimal value around 64 threads, this causes the overall runtime to be higher than the optimal fixed-size pool. While the pool size in phase 1 and 3 does not stabilize around the optimal of 16, the pool is not overscaled to a size over 35 and only at the beginning of phase 1 and shortly after again falls below 10 threads. In chapter 4 figure 4.1 it has been shown that between 10 and 35 threads the runtime is roughly the same and close to optimal for the unbuffered read-write jobs, so the adaptive pool behavior for phase 1 and 3 does not affect runtime negatively.

Overall for this multi-phase workload the adaptive pool performs relatively close to the optimal fixed-size pool, but it fails to correctly adapt to phase 2. The potential to outperform the optimal fixed-size pool is therefore not reached.

Unsaturated-saturated The second type of phase change, besides a change in job type, is a change in the job submission rate. We devised a workload that consists of a first phase of medium load that does not saturate the thread pool's job queue and a second phase of instant submission of a big batch of jobs which saturate the thread pool's job queue. The job type is the same for phase, it's the non

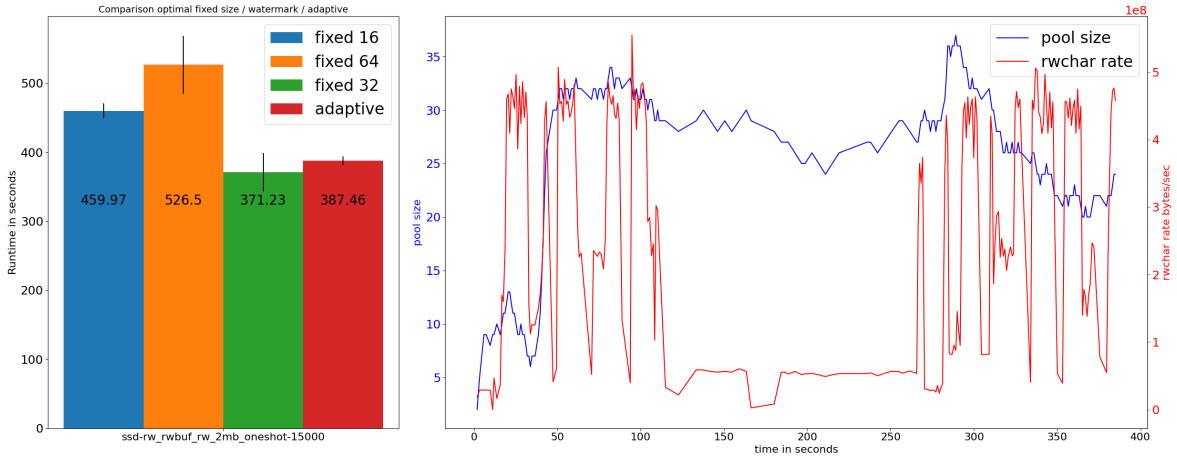


Figure 5.3: Multi-phase unbuffered-buffered performance/behavior

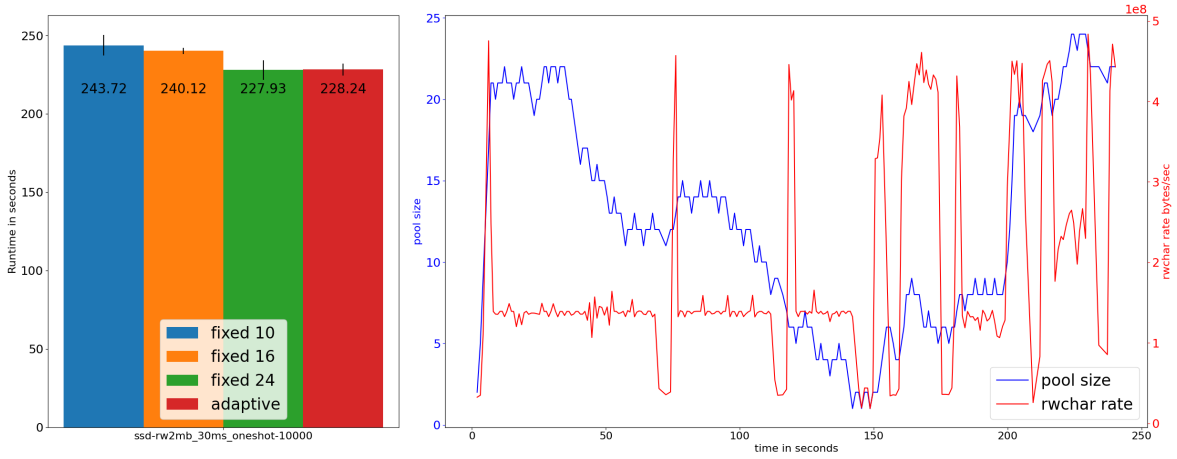


Figure 5.4: Multi-phase unsaturated-saturated performance/behavior

buffered read-write job from the previous section. In the first phase 5000 jobs are submitted with a frequency of 30 milliseconds, after which 5000 jobs are submitted immediately to mark the start of the second phase.

The optimal fixed-size pools for the respective phases and the whole multiphase workloads are 10 (for the unsaturated phase), 16 (for the saturated phase) and 24 (for whole workload). The runtimes are compared with the runtime of the adaptive pool in the left graph of figure 5.4. The adaptive pool achieves a runtime as low as the optimal fixed-size pool (24 threads). The right graph of figure 5.4 again shows the scale behavior of the adaptive pool during the execution of the workloads. From the *rwchar* rate one can distinguish the 2 phases of the workload, with the disk throughput being mostly constant throughput the first phase (first 150 seconds). During this time the pool is first scaled up to about 25 threads and then continuously scaled down, as the throughput stays constant. Once the extra batch of jobs is submitted to start the second phase, the throughput peaks for longer periods and fluctuates more. The pool is scaled back up again, reaching a size of around 25 at the end of execution.

5.3.3. NodeJS

Implementation We use Node version 14.5.1. Node uses a fixed size thread pool implemented by the Libuv project to run disk I/O related operations [26]. The pool size can be set at startup through the `UV_THREADPOOL_SIZE` environment variable and is by default set to 4. The thread pool architecture is similar to the adaptive pool we implemented, so we could integrate the scale adapter in the same fashion as described in the previous chapter. The worker threads always trigger the scale advice

method before consuming the next job from the work queue and in case of a non-zero scale advice, terminate or clone commands are pushed to the front of the work queue to be consumed by the next workers that are ready.

Single-phase Workload Experiments In the following we take a look at the experimental results for the previously shown workload rw_sync_10mb-node and a second read-write workload that does not perform manual `fsync` on the written file (rw_nosync_2mb-node). We compare total runtimes and average pool size for the default fixed-size pool, the optimal fixed-size pool and adaptive pool. The scale adapter was initialized with an interval length of 1500ms and a stability factor of 0.9.

Figure 5.5 shows a comparison of average runtimes and average pool sizes of the 3 different thread pools for the 2 read-write workloads. The optimal fixed-size pool is shown by the blue barplots, the default fixed-size pool by the orange ones and the adaptive pool by the green ones. In the top graph we see that for both workloads the default fixed-size pool does only perform slightly worse than the optimal fixed-size pool, being about up to 10% slower. The adaptive pool is up to 25% slower than the default pool. In the bottom graph we see a first indicator why the adaptive pool fails to beat even the default pool, its average pool size is lower than the default pools 4 for both workloads.

In order to understand this behavior we investigate the adaptive pool behavior during the execution of both workloads. Figure 5.6 shows the *rwchar* rate and pool size over time. The top graph shows the workload with 10Mb manually synced files, the bottom graph shows the workload with 2Mb files not manually synced. For both workloads the initial scaling behavior is very similar, the adapter scales up the pool to a size of 9-10, then almost as rapidly scales the pool down back to a size of 1. For the synced workload the pool size oscillates between 1 and 2 from then on, before increasing up to 6 in the last 30 seconds of execution. During the non-synced workload the pool size fluctuates between 2 and 5 with a 40 second period of consistent pool size 1 in the middle of execution.

The *rwchar* rate fluctuates heavily with regular interleaved peaks at roughly every second interval. This behavior persists throughput the whole execution of both workloads, with the synced workload showing peaks that more than double the *rwchar* rate of the previous interval and the other workload showing peaks with usually about 30% increases of *rwchar* compared to the previous interval.

The regular pattern of significant interleaves increase and decrease in *rwchar* rate causes the scale adapter to fail to correctly adapt the pool size. As the feedback is restricted to only the previous interval, the algorithm has no mechanism to avoid erroneously reacting to the fluctuations. In the next chapter we introduce adjustments to the algorithm that partly alleviate the influence of these kind of regular flucutations on overall scaling behavior.

5.3.4. RocksDB

We use RocksDB version 6.7.3. RocksDB uses a thread pool solely for synchronizing data files to the disk in order to guarantee durability, which is called the flush pool. The flush pool uses the Watermark scheme with a minimum pool size of 1 and a recommended maximum pool size of 2 [1]. Instead of rewriting the pool completely to match our proposed architecture, we make use of the already existing functionality that can change the maximum pool size at runtime.

Implementation - No-manager Worker threads trigger a scale advice before consuming the next job from the work queue and on receiving a non-zero scale advice they directly change the maximum pool size to match the scaling advice. Then the existing method to spawn/terminate workers based on the current maximum pool size is called before the worker moves on to the actual job.

As we will show in figure 5.8, due to the job execution times being much longer than the scale advice intervals, the scale advice method is only triggered much less frequently than desired when just called from the worker threads. In order to regularly trigger the adaption procedure we implemented an alternative integration of the scaling adapter into the RocksDB thread pool.

Alternative Implementation - Manager Thread In this architecture an extra thread that takes the sole responsibility of regularly calling the scale advice method and adjusting the pool size accordingly is spawned. This manager thread just sleeps for the specified interval length, then obtains scale advice and in case of non-zero advice triggers a rescaling through the already existing mechanism that spawns

[1] https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning

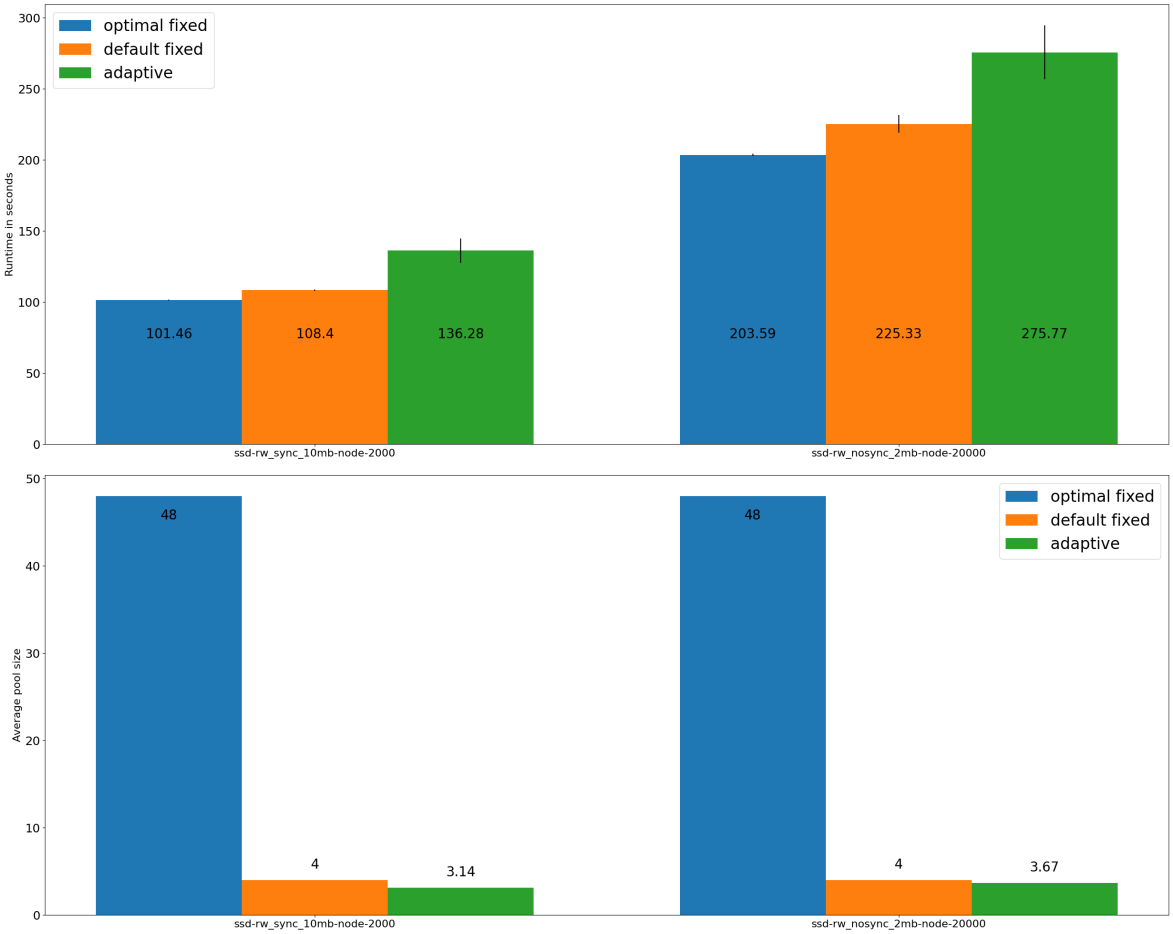


Figure 5.5: Node - Comparison of default and optimal fixed size, adaptive pool

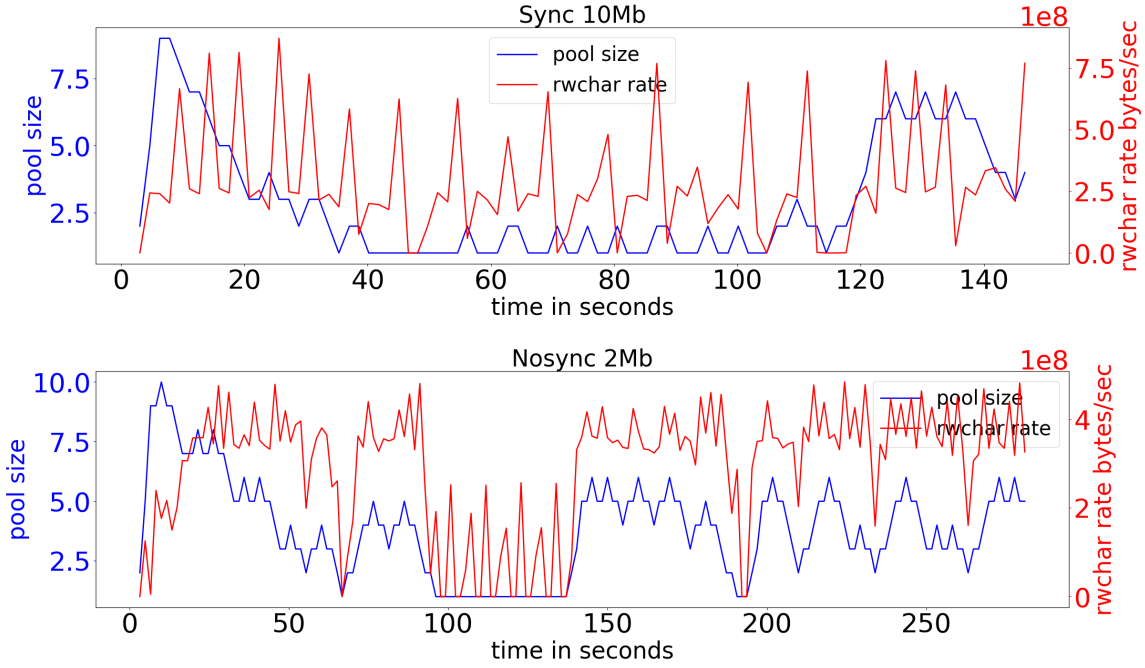


Figure 5.6: Node - Adaptive pool behavior during execution

new workers or instructs existing workers to terminate. As busy workers can not reasonably terminated during job execution, scale down actions take considerably longer than scale up actions

Single-phase Workload Experiments We compare both implementations with the recommended default fixed-size flush pool and the experimentally determined optimal fixed-size pool. The adapter algorithm was instantiated with a interval length of 1000ms and a stability factor of 0.9, which was determined to perform better than other tested parameters. Both workloads are write-heavy batch insertion workloads. Figure 5.7 shows a runtime comparison in the top graph and a comparison of average pool sizes for the 4 different configurations.

The default configuration of a flush pool with maximum size 2 performs the worst, but the optimal fixed-size configuration with 12 threads significantly outperforms the two adaptive implementations for both workloads.

Fillseq The no-manager implementation's runtime is more than 20% slower, the manager implementation's runtime is about 15% slower. Just looking at the average pool size it is clear that the no-manager implementation underscales the pool size by a large margin. The manager implementation's average pool size close to the optimal fixed size, but it still performs much worse. When inspecting the runtime behavior we see that while the manager implementation may have a good pool size on average, it does vary too much during the execution.

Bulkload The two adaptive implementations achieve the same runtime, with the manager implementation having a lower average pool size, which presumably causes it not to outperform the no-manager implementation.

Figure 5.8 shows the pool size and *rwchar* rate over time when using the adaptive pool for the fillseq workload. The top graph shows the no-manager implementation, the bottom graph shows the manager implementation. First of all, it is clear that through deferring the responsibility to get scale advice to a manager thread the intervals between scale actions can be shortened. However, this does not hold for some intervals that follow a scale down action, it may take a long time for any worker to complete its job and is able to perform a scale down by terminating itself. The adapter does not obtain new scale advice until the scale actions have been performed as it would introduce even larger lag between issuing and performing of scale actions when scaling down. The full effect of a previous rescaling of the pool size should be observed before determining new scale actions, otherwise the associated effect of an action on the target metric is not captured. So even with the manager approach long job execution times remain problematic.

The manager implementation does scale up the pool significantly more for this workload, a peak size of 20 is reached, whereas the no-manager implementation peaks at a pool size of 8.

5.4. Discussion

The most challenging and crucial aspect of an OS-feedback based pool size adaption scheme is the OS metric or the OS metrics that are used as performance indicators. For workloads restricted to specific jobs, such as the disk I/O-bound workloads we investigated, it is possible to use the *rwchar* rate that is highly correlated with overall throughput. However, a highly correlated metric such as the *rwchar* rate may still be fluctuating heavily. This is problematic for a controller-based approach, which is intrinsically sensitive to fluctuations in the target metric.

Overall we see that the adaptive thread pool can approach optimal performance for single-phase workloads when the target performance metric is relatively stable, i.e does not fluctuate heavily. As soon as the *rwchar* rate shows significant and steady fluctuation, the scaling adapter fails to correctly scale the pool towards an optimal size, as can be seen in the Node experiments. In the following chapter we show how we adjusted the scaling algorithm to partially compensate for heavy fluctuations and improve the scale behavior.

For the tested multi-phase workloads on our adaptive pool implementation the total runtimes are similar to the optimal fixed-pool case, but did not reach the possible lower bound. The cause for this also lies in the periods of instability of the *rwchar* rate, these are the periods during which the pool scales down too much.

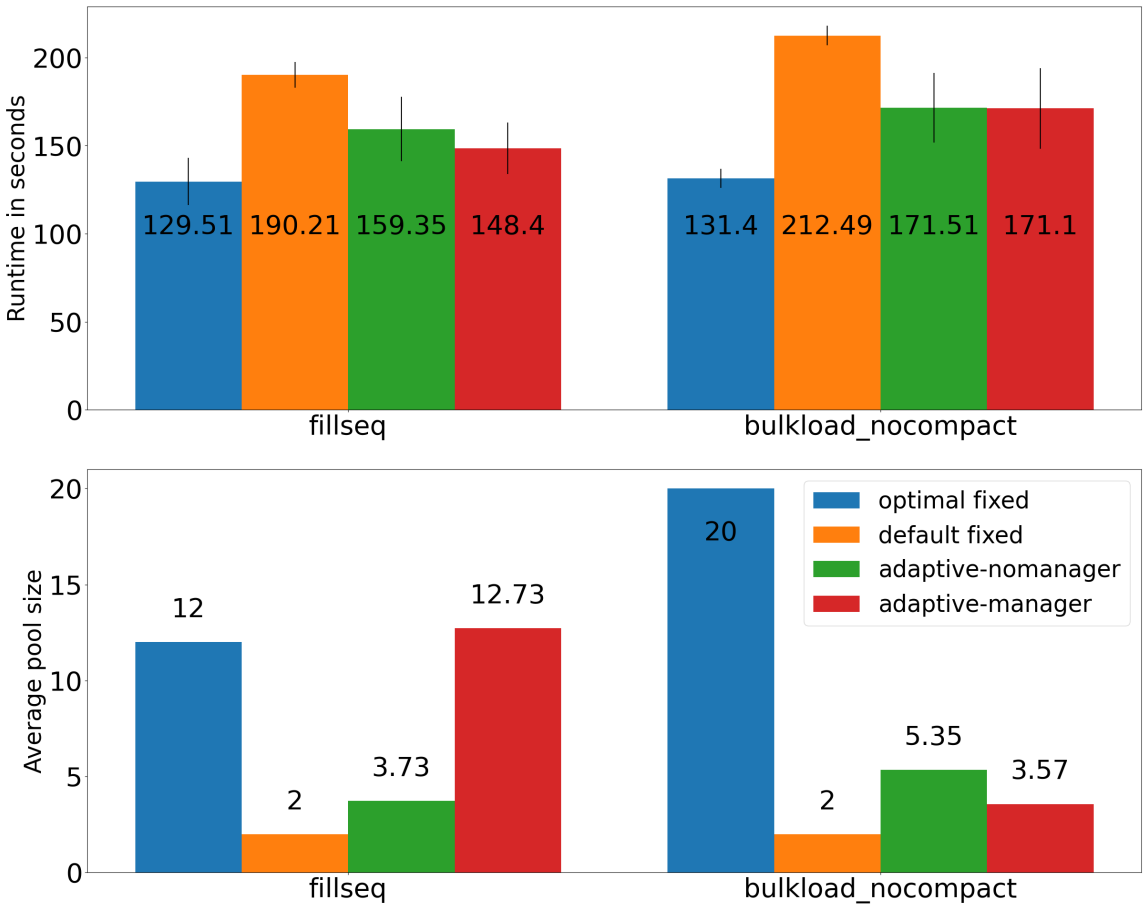


Figure 5.7: RocksDB workloads - performance comparison

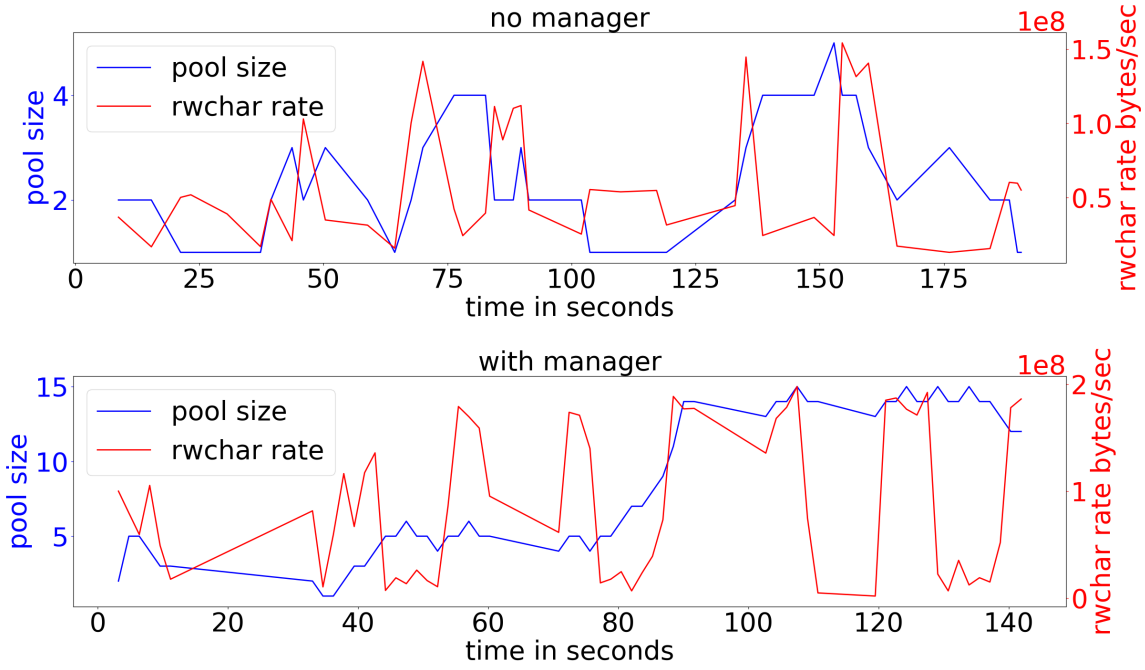


Figure 5.8: Fillseq - adaptive pool behavior during execution

A second issue besides an instable target metric arises when the job execution times exceed the interval parameter for the scale adapter. This is more problematic with the proposed architecture of worker threads themselves being responsible for regularly obtaining scale advice and issuing scale/terminate commands. The alternative to this, an extra manager thread, was introduced for the RocksDB integration in the previous section and alleviates this problem to some extent.

In the next chapter we introduce two extensions to the algorithm that aim to reduce the impact of fluctuations in the target metric on the scaling behavior. We'll also discuss the results and limitations of our approach on a more fundamental level.

6

Algorithm Extensions & Analysis

In this chapter we propose some extensions to the adapter algorithm that address the instability of the *rwchar* rate that occurs for some workloads. Specifically we focused on improving the performance of the adaptive pool for the Node workloads, where we found the heavy fluctuations to impact scale behavior the most. We introduce two extensions to the algorithm that mitigated the impact of the fluctuating *rwchar* rate to some extent. The extensions and their motivation are first described, then their performance is again experimentally evaluated for the Node and RocksDB workloads. In the last section we then summarize the performance of these extensions for all other workloads from the last chapter and discuss the extension's applicability and limitations to a wider range of workloads.

6.1. Algorithm Extensions

6.1.1. Drop Exception Rule

We introduce a simple heuristic rule that aims at reducing interference of large fluctuations in the logical disk throughput. When the scaling adapter is in one of the Settled, Exploring or Scaling states and the *rwchar* rate suddenly drops by at least 70%, the current state is unchanged and no scale actions are issued. This factor of 70% was chosen to capture the drops as observed for the Node workloads, further testing of different factors was out of scope for our purpose. With this heuristic we can prevent single downward fluctuations to interrupt the Settled, Exploring and Scaling phases, effectively allowing the adapter to ignore single intervals with large throughput drops. The heuristic aims to stabilize the scaling behavior and facilitate scaling the pool to a sufficiently large size.

In table 6.1 we compare the extended algorithm's performance to default and optimal fixed-size as well as the unmodified algorithm. For the Node workloads an interval length of 800ms with a stability factor of 0.97 was used, for the RocksDB workloads as interval length of 1000ms and a stability factor of 0.9 was used. The table reports average total runtimes and average pool sizes in parentheses. Standard deviations of runtime were omitted, they were lower for the extended algorithm compared to the unmodified version without exception. Column "Drop. Ex" reports the performance for the *drop exception* extension and "Mov. Avg." reports the performance for the *moving average* extension introduced next.

With the *drop exception* extension total runtimes for all workloads could be reduced significantly, average pool sizes are larger except for the RocksDB Fillseq workload. However, for the Node workloads the default configuration with 4 worker threads still outperforms the adaptive pool. The extended

Workload	Default fixed	Optimal fixed	Adaptive	Drop Ex.	Mov. Avg.
RW-sync	108.4s (4)	101.5s (48)	120.3s (2.9)	114.7s (6.2)	116.8s (8.1)
RW-nosync	225.3s (4)	203.6s (48)	255.0s (3.8)	246.7s (5.7)	229.2s (14.3)
Fillseq	190.2s (2)	129.5s (12)	148.4s (12.7)	130.8s (6.9)	137.6s (12.3)
Bulkload	212.5s (2)	131.4s (20)	171.1s (3.6)	141.6s (7.5)	137.5s (21.2)

Table 6.1: Comparison avg. runtimes and avg. pool sizes - Node/RocksDB

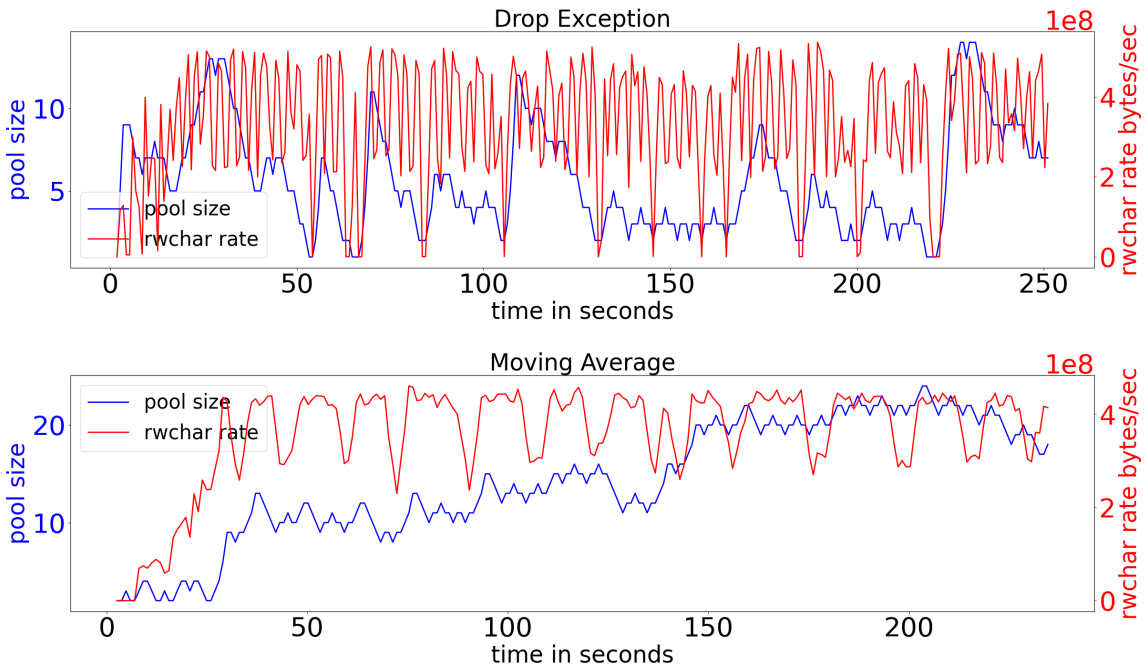


Figure 6.1: Adapter behavior - rw_nosync_2mb-node

adaptive pool performs better for the RocksDB workloads, approaching optimal performance with a 10% margin for the Bulkload workload and less than 1% slower than optimal for the Fillseq workload.

6.1.2. Moving Average

Another approach to reducing the fluctuations of *rwchar* rate from one interval to the next is to extend the interval lengths. However with the previous algorithm the interval length also determines the frequency of scaling actions, extending it would make the scaling process much slower. Therefore we introduced an extra parameter (*averaging_duration*) to decouple the interval length over which the *rwchar* rate is computed and the interval length that determines frequency of scaling actions (*scale_interval*). By choosing an averaging duration larger than the scale interval we hope to smoothen the *rwchar* rate curve over time while keeping the frequency of scaling actions high. For the Node workloads a scale interval of 1000ms with a stability factor of 0.95 and an averaging duration of 5000ms was used, for the RocksDB workloads a scale interval of 1000ms a stability factor of 0.9 and an averaging duration of 5000ms was used.

The results are also reported in the table 6.1 in the last column. Compared to the *drop exception* extension, the *moving average* extension performs significantly better only for the Node Nosync workload, for other workloads total runtime is maximum 5% larger or smaller. Average pool sizes are significantly larger for all pool workloads, but don't significantly exceed the optimal pool sizes for any workload. Regardless of fluctuations in pool size, the adapter underscales the thread pool for both Node workloads with any variation on the adapter algorithm.

6.1.3. Analysis & Evaluation

Figure 6.1 shows the adapter scaling behavior for both extensions for the Nosync Node workload. With the *drop exception* extension the adapter repeatedly scales the pool up and down alternating roughly between pool size from 10 to 3. The *moving average* extension shows completely different behavior, the pool scaled up quickly around 30 seconds and 140 seconds into the execution. The remainder of the time the pool size is relatively stable, resulting roughly 3 periods of stable pool sizes.

Figure 6.2 shows the adapter scaling behavior for both extensions for the Fillseq RocksDB workload. With the *drop exception* extension the adapter continuously scales up the pool throughput the workload, reaching a pool size of around 60 at the end. With the *moving average* extension the adapter first quickly

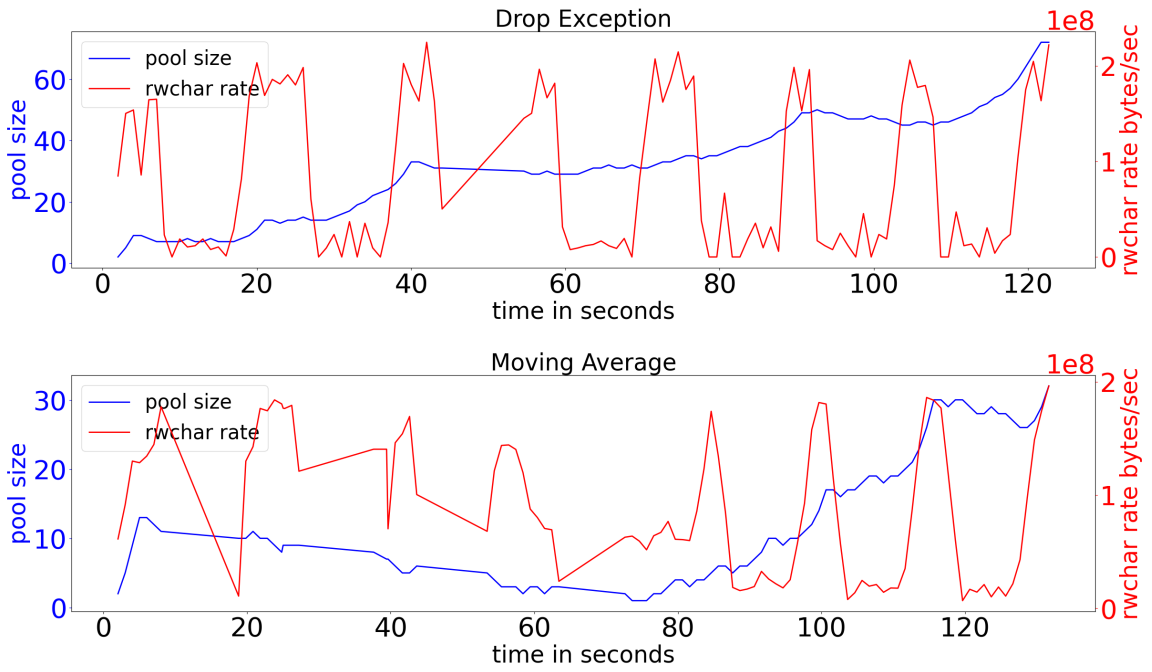


Figure 6.2: Adapter behavior - fillseq

Workload	Watermark	Optimal fixed	Adaptive	Drop Ex.	Mov. Avg.
read_2mb	130.1s (64)	128.2s (4)	140.7s (4.5)	138.6s (5.6)	130.1s (12.1)
rw_2mb	625.7s (64)	330.8s (16)	324.7s (16.6)	362.4s (18.0)	377.5s (16.3)
rw_buf_2mb	163.4s (64)	156.6s (64)	194.7s (31.4)	219.6s (36.6)	258.1s (23.0)
rw_rwbuf_rw_2mb	500.0s (64)	371.2s (32)	387.5s (25.3)	431.0s (22.3)	507.3s (16.5)
rw2mb_30ms_oneshot	297.8s (?)	227.9s (24)	228.2s (12.5)	233.8s (7.7)	248.1s (27.0)

Table 6.2: Comparison avg. runtimes and avg. pool sizes - Adaptive Thread Pool

scales the pool to about 15 threads and then over the next minute continuously scales down the pool. The the pool is scaled up to around 30 threads over almost the whole remainder of execution.

We also evaluated the two algorithm extensions for our own adaptive thread pool implementation and the more 'ideal' workloads (short jobs, more stable *rwchar* rate). The results are shown in table 6.2, which again reports average runtimes and average pool sizes in parentheses. The unmodified adaptive pool is instantiated with a 800ms scale interval and a stability factor of 0.97. The *drop exception* adapter with 1500ms scale interval and 0.95 stability factor was used. The *moving average* adapter with 1500ms scale interval, 0.95 stability factor and 3000ms averaging duration was used.

With the exception of the read_2mb workload, where both extended versions of the algorithm achieve a lower runtime, the modified algorithms perform consistently worse than the original version. This shows that the extensions are an over-optimization for the Node and RocksDB workloads and not generalizable. This is not entirely surprising as they are aiming to reduce downward scale actions for some specific workloads to counteract fluctuations. A better approach would be to develop a more sophisticated target metric as we discuss in the next section.

6.2. Discussion

The two introduced algorithm extensions do improve performance overall for the workloads tested for Node and RocksDB at the cost of higher average thread pool sizes. However, as we have shown in figure 6.1 and figure 6.2 the scaling behavior of both variations is quite instable, the pool size is changed significantly throughput the whole workloads, as opposed to settling on some size once and then only

adapting it slightly from time to time as observed in the synthetic read-write workloads (figure 5.2). We think this is an intrinsic limitation of the chosen target metric, more sophisticated target metrics to maximize need to be developed to fundamentally solve this issue. Apart from this, jobs submitted to the thread pool that take too long to execute are problematic regardless of the used target metric, they interfere with the ability of the scale adapter to timely react to changes in the target metric.

A more effective target metric than the *rwchar* rate, which captures current throughput, would be predictive of expected throughput and possible contention, so the controller can reduce pool size preemptively to avoid throughput drops as a result of disk contention. Alternatively a second metric that captures contention may be used to detect the onset thereof, triggering the controller to scale down once a specific threshold is reached. We did experiment with some metrics related to system call times and disk blocking times (iowait) to implement the second approach, but ultimately did not find a sufficiently generalizable metric within time.

7

Conclusions & Future Work

In this chapter we summarize our contributions and discuss the results of this research w.r.t to the initial research questions. Furthermore, we reflect on the limitations of our proposed solution and propose directions for future research that may address them.

7.1. Conclusions

First of all, in chapter **4** we have shown that there is an obvious OS metric that shows high correlation with total runtime of disk I/O workloads, namely the logical disk throughput as represented by the *rwchar* rate, addressing our **research question 1**. Some investigation into alternative or additional metrics were made, like the I/O syscall rate reported in section 4.1.3 and others that were omitted. In the end we concentrated just on the *rwchar* rate for our adaptive thread pool implementation with good results for the tested workloads, but we believe this is not the only OS metric that can be used for a similar feedback-based adaptive algorithm and it may be of benefit to combine multiple metrics to lessen the impact of instability in a single metric (more on this in the coming section 7.2).

We introduced our algorithmic approach that is based on maximizing the *rwchar* rate through an extended hillclimbing approach which is encapsulated in a controller module, the scale adapter. The scale adapter makes use of 3 principal internal states; Scaling, Settled and Exploring to continuously adapt the thread pool size. We discussed the architecture of our implementation; the mechanism for tracing metrics (traceset), the controller module (scale adapter) and the integration thereof into a thread pool. Finally we addressed our **research question 2a** by testing the overhead created of regularly calling the adapter's scale advice method. We found the overhead to be unobservable w.r.t total runtime and thus negligible.

In chapter **5** we defined a simple model to classify the workloads that were used in the following experimental analysis and we argued that our approach is most suitable for the kinds of workloads that are characterized by relatively stable phases. We showed that for read-write workloads that fit this criteria our adaptive thread pool implementation shows good performance w.r.t the goal of minimizing both total runtime and the average amount of threads used. The results in comparison to the respective optimally sized fixed pools support the conclusion that for single-phase workloads the proposed adaptive solution can approach near-optimal performance, thus answering our **research question 2b**.

In order to address **research question 2c** our solution was also compared to the common Watermark scheme, which we could demonstrate to perform significantly worse w.r.t total runtime for some workloads and significantly worse w.r.t amount of threads used for the other workloads while having comparable total runtimes.

We tested the ability of the scale adapter to detect and readjust pool size for 2 dynamic multi-phase workloads. While performing close to the optimally sized fixed-size pool and scaling the pool differently in the different phases, the adaptive solution clearly did not outperform it, failing to reach potential optimal performance. Thus w.r.t **research question 2d** we could not fully realize the original intended goal.

The scale adapter was also integrated into two applications, Node and RocksDB, with the experimental analysis revealing two limitations of the proposed solution. Firstly, the *rwchar* rate proved to

be too instable for the 2 read-write workloads that were tested, resulting in poor performance due to the scale adapter's constant switching of states because of the heavy, regular fluctuations. Secondly, for the write-heavy RocksDB workloads the internal flush pool was assigned jobs with long execution times, that made regular scale action impossible and thus reduced the ability of the scale adapter to quickly adapt pool size.

In chapter **6** we introduced two extensions to the core scaling algorithm. These extensions improved performance for the Node and RocksDB workloads, but do not completely alleviate the issue of a fluctuating *rwchar* rate and long job execution times. Instead we conclude that more sophisticated target metrics are needed that are predictive of resource contention instead of descriptive only.

Overall our work serves as an exploration of the solution space for feedback-based thread pool size adaption schemes that use OS metrics as opposed to system-agnostic metrics such as job completion rates. We believe to have shown this alternative approach to be feasible in the context of restricted workloads that mainly use one kind of system resource such as disk I/O bandwidth. Further testing in different kind of application contexts and exploration of more OS metrics are needed and in the next section we argue that a global (system-wide) approach should be considered.

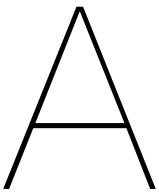
7.2. Future Work

First of all we think that our approach could be improved through making use of additional metrics that the OS provides, such as metrics exposed by the disk devices, and metrics that have to be obtaining through tracing, such as call durations and frequencies of system calls. The benefits of a more sophisticated target metric to maximize may be an improvement in stability which in turn makes simple hillclimbing more effective. Furthermore, through the addition of a complementary metric predicting disk contention (instead of observing it through a throughput metric) the scale adapter could avoid frequent suboptimal rescaling due to fluctuating throughputs in periods of contention.

As we have pointed out earlier, there are currently many asynchronous runtimes that use thread pools for disk I/O only workloads where our approach could be further tested and refined. It is not clear if it is possible to develop a general adaption algorithm that performs well for a majority of integrations into different applications, our results suggest that the same adapter algorithm's performance can vary significantly for different application's workloads.

From a broader perspective, we believe that a adaptive approach to determine concurrency levels of applications should ultimately be globally performed within the kernel, as scheduling of jobs and adjustment of concurrency levels can be performed according to resource usage of a job and current resource utilization if done within the kernel. The scale adapter would have the most complete knowledge over system-wide outstanding jobs and system resources to make these decisions. Instead of one or more thread pools per application, an in-kernel implementation could be split in one thread pool per resource, such as CPU, disk device and network device. Jobs at the granularity of single system calls can be executed on the specific resource pool according to the characteristic resource usage of the system call. Using in-kernel global thread pools instead of per-application pools should additionally be beneficial for performance isolation between multiple concurrent applications and provide better control over QoS (Quality of Service) requirements.

The recently introduced io_uring interface to Linux offers a mechanism for submission of asynchronous system calls and collection of results, so there is already infrastructure in place for a kernel-side implementation that is exposed to the user at a system call granularity [1, 5]. Io_uring is already used in concurrency libraries such as Glommio to replace dedicated I/O thread pools, shifting the responsibility of concurrent execution to the kernel, where an adaptive scheme to determine concurrency levels would benefit all user applications [6]. With the proliferation of asynchronous programming and asynchronous system architectures lies the opportunity to completely relieve the application programmer of the burden to make explicit use of or to have to tune thread pools. Nevertheless, concurrency still has to be managed within the kernel, where we believe an adaptive scheme to be very appropriate.



Appendix

A.1. Workloads

In the following we list all the workloads that were used for experimental analysis along with a short description of them.

A.1.1. Adaptive Thread Pool

- read_2mb: One job consists of reading an uncached 2Mb file from disk. All jobs are submitted at once at the start.
- rw_2mb: One job consists of reading an uncached 2Mb file from disk fully into memory and then writing the contents to a new file on disk. The file is then synchronized with the `fsync` system call before being removed. All jobs are submitted at once at the start.
- rw_buf_2mb: One jobs consists of reading an uncached 2Mb file from disk in chunks of 4Kb and writing the contents chunk per chunk to a new file. After every completed write of a chunk the file is synchronized with the `fsync` system call. When the whole file has been written it is removed. All jobs are submitted at once at the start.
- rw2mb_30ms_oneshot: The jobs are the same as in the rw_2mb workload. In the first phase jobs are submitted at a rate of one job every 30ms. In the second phase a batch of jobs is submitted all at once.
- rw_rwbuf_rw_2mb: A first set of jobs from the rw_2mb workload is submitted, a second set of jobs from the rw_buf_2mb workload is submitted, a third set of jobs from the rw_2mb workload is submitted. All jobs are submitted at once at the start. The jobs are executed in order of submission, so the whole execution is effectively split in 3 different phases.

A.1.2. NodeJS

The workloads are regular Node scripts that execute functions in parallel through the `Promise.all` function. The functions are executed in batches of 100.

- rw_sync_10mb: The function consists of reading an uncached 10Mb file from disk fully into memory and then writing the contents to a new file on disk. The file is then synchronized with the `fsync` system call before being removed.
- rw_nosync_2mb: The function consists of reading an uncached 2Mb file from disk fully into memory and then writing the contents to a new file on disk. The file is then removed.

A.1.3. RocksDB

Both workloads are write-heavy workloads that write new objects to the key-value store. The jobs executed by the flush thread pool are batches of `fsync` operations on sets of data files.

- fillseq: writes N values in sequential key order, no reads are performed
- bulkload: writes N values in random key order, no reads are performed

A.2. Chapter 6 omitted results

Workload	Default fixed	Optimal fixed	Adaptive	Drop Ex.	Mov. Avg.
RW-sync	0.6	0.3	6.0	3.3	4.8
RW-nosync	6.2	0.7	8.9	5.0	2.4
Fillseq	7.3	13.4	14.6	7.5	10.9
Bulkload	5.5	5.3	23.0	1.8	3.7

Table A.1: Comparison runtime standard deviations - Node/RocksDB

Workload	Watermark	Optimal fixed	Adaptive	Drop Ex.	Mov. Avg.
read_2mb	0.1	0.5	8.8	7.3	0.1
rw_2mb	53.7	29.8	47.9	35.1	30.9
rw_buf_2mb	14.9	10.1	25.4	25.1	25.8
rw_rwbuf_rw_2mb	56.2	27.8	6.1	5.3	43.4
rw2mb_30ms_oneshot	17.8	6.2	3.7	7.2	12.6

Table A.2: Comparison runtime standard deviations - Adaptive Thread Pool

A.3. Source Code

The source code for the implementation of the scaling adapter and the modified applications, as well as scripts used for benchmarking can be found on the following Github profile: `https://github.com/jannes-thesis`.

Bibliography

[1] Jens Axboe. Efficient io with io_uring, 2019. URL https://kernel.dk/io_uring.pdf. [Online; accessed 3. Feb. 2021].

[2] John Calcote. Thread pools and server performance-as john points out here, thread pools provide one way of improving server performance. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 22(7):60–65, 1997.

[3] Ning-jiang Chen and Pan Lin. A dynamic adjustment mechanism with heuristic for thread pool in middleware. In *2010 Third International Joint Conference on Computational Science and Optimization*, volume 1, pages 369–372. IEEE, 2010.

[4] Jonathan Corbet. Toward non-blocking asynchronous I/O [LWN.net], 2017. URL https://lwn.net/Articles/724198. [Online; accessed 8. Feb. 2021].

[5] Jonathon Corbet. The rapid growth of io_uring [lwn.net], 2020. URL https://lwn.net/Articles/810414. [Online; accessed 3. Feb. 2021].

[6] Glauber Costa. Glommio, 2020. URL https://github.com/DataDog/glommio. [Online; accessed 9. Dec. 2020].

[7] Nilushan Costa, Malith Jayasinghey, Ajantha Atukoralez, Supun Abeysinghex, Srinath Perera, and Isuru Pererak. Adapt-t: An adaptive algorithm for auto-tuning worker thread pool size in application servers. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2019.

[8] Jake Edge. Unifying kernel tracing [LWN.net], 2019. URL https://lwn.net/Articles/803347. [Online; accessed 22. Oct. 2020].

[9] Facebook. RocksDB benchmarking tools, Jan 2021. URL https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. [Online; accessed 15. Jan. 2021].

[10] Facebook. RocksDB, 2021. URL https://github.com/facebook/rocksdb. [Online; accessed 4. Jan. 2021].

[11] Brian Goetz, Tim Peierls, Doug Lea, Joshua Bloch, Joseph Bowbeer, and David Holmes. *Java concurrency in practice*. Pearson Education, 2006.

[12] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[13] Brendan Gregg. Choosing a Linux Tracer (2015), Jul 2015. URL http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html. [Online; accessed 22. Oct. 2020].

[14] Joseph L Hellerstein, Vance Morrison, and Eric Eilebrecht. Optimizing concurrency levels in the. net threadpool: A case study of controller design and implementation. *Feedback Control Implementation and Design in Computing Systems and Networks*, 2008.

[15] DongHyun Kang, Saeyoung Han, SeoHee Yoo, and Sungyong Park. Prediction-based dynamic thread pool scheme for efficient resource usage. In *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 159–164. IEEE, 2008.

[16] Dan Kegel. The c10k problem, 2006.

[17] Michael Kerrisk. proc(5) - Linux manual page, Dec 2020. URL https://man7.org/linux/man-pages/man5/proc.5.html. [Online; accessed 4. Jan. 2021].

[18] Sobhan Omranian Khorasani, Jan S Rellermeyer, and Dick Epema. Self-adaptive executors for big data processing. In *Proceedings of the 20th International Middleware Conference*, pages 176–188, 2019.

[19] Ji Hoon Kim, Seungwok Han, Hyun Ko, and Hee Yong Youn. Prediction-based dynamic thread pool management of agent platform for ubiquitous computing. In *International Conference on Ubiquitous Intelligence and Computing*, pages 1098–1107. Springer, 2007.

[20] Kang-Lyul Lee, Hong Nhat Pham, Hee-seong Kim, Hee Yong Youn, and Ohyoung Song. A novel predictive and self–adaptive dynamic thread pool management. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pages 93–98. IEEE, 2011.

[21] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review*, 34(2):42–55, 2000.

[22] Oracle. ThreadPoolExecutor (Java Platform SE 8), Jul 2020. URL `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html`. [Online; accessed 23. Nov. 2020].

[23] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64. Citeseer, 2005.

[24] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[25] Douglas C Schmidt. Evaluating architectures for multithreaded object request brokers. *Communications of the ACM*, 41(10):54–60, 1998.

[26] The Libuv maintainers. Libuv, 2020. URL `https://github.com/libuv/libuv`. [Online; accessed 7. Dec. 2020].

[27] The Linux maintainers. Linux Tracing Technologies - The Linux Kernel documentation, 2021. URL `https://www.kernel.org/doc/html/latest/trace`. [Online; accessed 4. Feb. 2021].

[28] The MariaDB maintainers. Thread Pool in MariaDB, 2021. URL `https://mariadb.com/kb/en/thread-pool-in-mariadb/#thread-pool-internals`. [Online; accessed 4. Feb. 2021].

[29] The Node.js maintainers. Node.js, 2021. URL `https://github.com/nodejs/node`. [Online; accessed 4. Jan. 2021].

[30] The Tokio maintainers. Tokio, 2020. URL `https://github.com/tokio-rs/tokio`. [Online; accessed 7. Dec. 2020].

[31] The ZIO maintainers. ZIO, 2020. URL `https://github.com/zio/zio`. [Online; accessed 7. Dec. 2020].

[32] Dongping Xu. Performance study and dynamic optimization design for thread pool systems. Technical report, Ames Lab., Ames, IA (United States), 2004.