ADAPTIVE THREAD POOL SCALING AND REAL-TIME THREAD MONITORING

ADAPTIVE THREAD POOL SCALING AND REAL-TIME THREAD MONITORING

Author:
M.G.A. Janssen

Supervisor:
Prof.dr. J.S. Rellermeyer

A thesis submitted in
fulfillment of the requirements
for the degree of Master of Science
in the Distributed Systems Group
at the faculty EEMCS
of the Delft University of Technology

Student number:	4361709	
Defence:	15-08-2022	
Thesis committee:	Prof.dr.ir. D.H.J. Epema	TU Delft, chair
	Prof.dr. J.S. Rellermeyer	TU Delft, supervisor
	Dr.ir. Z. Al-Ars	TU Delft

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

Contents

ABSTRACT

Thread pools, integrated in programming languages, packages and dependencies are widely used by developers. Thread pools assume they are running alone on the system, which is not always the case. Previous research has shown that adapting thread pool size has been effective under specific conditions. In this research, scaling the thread pool with respect to CPU and network usage is examined. However, simpler metrics can achieve better results. Two solutions are provided for algorithmically scaling the thread pool, both with a different use-case in mind. Next to that an improved way to collect CPU and network metrics is provided, allowing for real-time thread-based measurements. The result of the scaling solution is improved performance, while also offering reduced energy usage. This research shows that when multiple thread pools are running on the same machine, performance and efficiency is improved.

PREFACE

With the master thesis being the last deliverable for obtaining the Master's degree in Computer Science at the Delft University of Technology, it marks the end of a period of learning and research. Learning will not end here, but this marks the completion of a solid foundation of knowledge which will be used to support further endeavours.

I would like to thank Prof.dr. J.S. Rellermeyer for his valuable advice, guidance and support during this process. Additionally, my gratitude for the members of the graduation committee, Prof.dr.ir. D.H.J. Epema and Dr.ir. Z. Al-Ars.

Finally, the people closest to me, my parents, sister and friends, thank you for all the support. Serving as a sounding board for ideas, offering support and feedback throughout the process.

Martijn Janssen
Delft, August 15, 2022

1

INTRODUCTION

1.1. OPENING

In present day enterprise computing, the environment has changed quite a bit. Previously, one application would be deployed on the server to fulfill the user's needs. Now the application is often composed of several smaller components. This move allows for a more even load distribution or better scaling due to the use of solutions like virtualization, containerization and colocation. These three solutions are deployed to clouds and datacenters, where they are used to run as many programs in parallel under the agreed upon SLA. Software provisioned to these servers is scaling up and down, reacting to the demand of the service. These software deployments have to share the available resources on the system with other programs running on the same system. The amount of applications with a microservice architecture are expected to increase by 18.6% each year from 2019 to 2026 [30]. Continuing on that trend, the cloud space is expected to achieve a 21.2% growth per year from 2019 to 2026. With these projected growth numbers for the future, improvements in performance or resource consumption are welcome.

The trend of sharing resources can be noticed in consumer computing as well. Personal computing usage nowadays often includes performing a video call, editing online documents, keeping shared folders synchronized, and listening to music through an online video platform. From a personal perspective, a selection of these programs can be running in the back- and foreground. This is not even taking into account the use of background processes by operating systems, such as mail client indexing, photo indexing, malware scanning, and a HDD defragmentation process.

Following this introduction, we take a more detailed look into previous developments in this area to provide further context. These developments can be split up into three major parts, CPU, disk and network.

1.1.1. CPU

While in previous hardware generations, performance increases have been delivered transparently through clock frequency increases (and turbo frequency increases), more modern chips cannot achieve this [19]. Instead, an increased number of independent compute cores are used [35]. This requires programs to use these independent compute cores differently and leverage their parallelism. Currently, CPU manufacturers are increasing core counts, marginally increasing clock speeds, and including the occasional hybrid architecture (a selection of performance- and lower clocked efficiency-cores) [28]. Years ago, the move to SMT (Simultaneous Multi-Threading), has shown a different path to achieve more performant software. By allowing two threads to run interchangeably on one core, a speedup of 1.20 for multiprogrammed workloads (independent threads) and 1.24 for a parallel workload (inter-thread communication) [3] was observed. With little to no hardware changes, it was possible to get improved performance, without making changes to clock speeds or try and make the program code itself more performant.

By incorporating mechanisms to take advantage of increased core counts and SMT into modern programming languages, increased program performance is made easier and more accessible. The important issue here is that these abstraction layers aimed at improving program performance behave as if they are the sole program running on the hardware. This is in reality not true, take the simplest example where the program is running twice on the same machine.

1.1.2. DISK

The next big component to consider for performance degradation in systems is disk performance. This is an area that was previously evaluated by Jannes Timm in his master's thesis [22]. The conclusion from that work was that actual solutions to the problems encountered were not effective, due to "heavy, regular fluctuations" in the measurements. The result was an improvement in performance, but the solution could be better if fluctuations could be resolved. The suspected issue here is that with how complex a system can get with regards to disk I/O, the parts like RAM, caching and the like make taking measurements more cumbersome or inaccurate.

1.1.3. NETWORK

The other major component where a bottleneck can be is in the network. While a process is reading and writing to the network, the network or the network interface can be at max throughput, thereby limiting performance. Compared to the disk, the network is a more simple component, lacking caching mechanisms which were found to be a problem in Jannes' thesis. Network messages are written to buffers, which in turn are written over the network. At a point during execution, these components will reach a limit and be the cause of degraded performance.

1.2. PROBLEM STATEMENT

As described above, there are several options for potentially improving performance by reducing contention in the system. This research is performed from the perspective of the Linux kernel, so statements apply to Linux OSs. To improve performance by reducing contention, we can formulate the main research question as:

RQ: Can we improve the performance of multiple programs on the same machine?

This main research question can be divided in several subquestions, which provides more structure for the reader:

RQ1: Can adaptive thread pools be extended beyond disk I/O?
As noted by Jannes Timm [22], performance for thread pools can be improved for disk I/O, but is this also the case for other resources?

RQ2: What is the most effective way of collecting metrics for the control loop to act upon?
Metrics collection is an important part of this solution. We need to ensure that the metrics that are used are accurate. Linux has numerous resources to collect metrics regarding running programs, but are they all equally good?

RQ3: Can the hillclimbing approach be improved to handle changes in variable workloads?
As observed in Jannes Timm's thesis [22], a hillclimbing method for scaling the thread pool is used which, finds the best value and sporadically explores nearby values after timeouts. Noted is that this hillclimbing approach is not effective for variable workloads. Can the hillclimbing algorithm be changed to perform better?

1.3. CHAPTER OVERVIEW

As an outline for this thesis, the answer to the main research question and following subquestions will be formed given the following steps. To begin, background information and previously completed work relevant to this research and topic is discussed in chapter 2. This chapter also evaluates available measurement tools and establishes requirements to improve upon current tools. Chapter 3 establishes the foundations for benchmarking and implementing a potential solution. In chapter 4, the monitoring setup is constructed, tested and validated. Once the monitoring is in working order, a rough analysis of the solution space is performed, confirming that there is room to improve performance. Chapter 5 proposes two universal solutions for improving performance using an algorithmic approach. These approaches are validated in chapter 6, which compares the two different solutions to the default. From the observations made, we follow with a conclusion in chapter 7 stating that the performance can be increased, while also having a positive effect on the power efficiency of programs. With all this information, we can also make a number of forward looking statements to serve as guidance for further research to be performed.

2

Overview

To have a system that is able to extract the best possible performance, there has to be a perfect cooperation between parts. This is still very much a utopia, as in practice, this is a difficult goal to achieve. The most important reason is that a current day system is a collection of abstractions, layers built on top of each other. One such an abstraction can force design decisions which will negatively impact the next layer, or the one it communicates with. In situations where it would make sense to solve a problem in a particular way, the result can be that a different application would have to interface with that solution in a suboptimal way. With all these smaller components being part of the bigger product, there is bound to be an area that can achieve better performance. Additionally, changes are being made every release cycle. If there are multiple incremental changes being made to the final product, it can be that there are no improvements to be made with those individual changes. However, once looking at the final product, there can be room for improvement in the future. This could be the case with regards to the research questions asked in the problem statement in section 1.2.

2.1. Related work

To get an overall view about the current state of multithreading, scheduling, usage of CPU and network resources, we need to look at previous research. In the following section, a structured overview is created for these subjects.

2.1.1. Concurrency

After the initial introduction of the Intel Pentium 4, which is the first processor available to the general public with the capabilities for SMT, Tuck et al. [3] evaluated the performance and how it was achieved. Concluded here is that there is a performance gain, but the limiting factor seems to be that the platform is not utilized entirely correctly, resulting in increased cache and resource conflicts. This issue was raised in 2003, but there have been core-count increases on top of SMT in the last few years for consumer hardware. Are there improvements to be made there?

Research performed by Dongping Xu [4] in 2004 showed an automated way to select the optimal amount of workers in a thread pool. The author notes: "In future applications, it is possible that the number of threads will become enormous. How will the method we have developed here scale in such cases?". While we are not directly using the method developed in that research, that remark is still valid. Does such a solution work nowadays, given all the changes that have been made to the operating system and hardware since then.

A fundamental part of a performant system with tons of programs running on it, is improved concurrency. This is addressed by Zhuravlev et al. [12], where a "comprehensive analysis of contention-mitigating techniques that use only scheduling" is produced. Their final solution after the analysis, which is a new scheduling algorithm, is able to perform within 2% of the optimal scheduling. This is achieved by applying a classification scheme that attempts to map contention for cache space and contention for shared resources (in this case memory controller, memory bus and prefetching hardware). However, they suggest that the biggest improvement with scheduling is not pure performance, but QoS and performance isolation. The resulting performance isolation achieves a speed up of 2x in select applications. This is a great end result to achieve, but this only applies to specific situations and does need prerequisite knowledge about the loads running on the machine, distinguishing between prioritized and non-prioritized applications.

The fact that resource sharing between concurrent applications is a difficult problem is touched upon by Hood et al. [11], where the result of this contention is evaluated in terms of a performance penalty. With memory as the subject, thread pool improvements are made by Jannes Timm [22], where it is shown that performance is improved by reducing memory contention by scaling the thread pool for a single application. In the same spirit, Khorasani et al. have reduced execution time of I/O intensive applications Terasort and PageRank with 34% and 54% respectively [21]. This is done by the use of an "adaptive executor" which is responsible for scaling the size of the thread pool in the Spark executor.

As an area of research that is more relevant today than it was in the past, there are possible opportunities to realize energy savings through the reduction of resource contention. This is suggested and shown by Porterfield et al. [7] [15] where performance and efficiency gains are realized through DVFS (dynamic voltage and frequency scaling) by throttling the processor during high power usage parts of execution where contention for a shared resource limits performance. This research shows real improvements, although validation is done with small benchmarks and the focus is solely on the CPU, not the entire system. With improvements like this, efficiency gains are possible through adjustments that have no impact on performance.

2.1.2. SCHEDULER

To allow concurrency in the CPU, the scheduler is responsible for allowing programs to take turns using the systems resources. A scheduler has a policy, which it uses to determine the order in which processes are given processing time. This policy can be chosen depending on the scheduler's use case. In Linux, the scheduler's job is mainly to find and divide the available processing time in a fair manner taking into account a per-process priority [32]. As noted by Wong et al. [8], programs can exploit Linux's CFS (Completely Fair Scheduler) and spawn more threads than they need. By doing this, these programs get more processing time than programs spawning less threads. This issue is resolved with the creation of a new scheduler, the TFPS (Thread Fair Preferential Scheduler). While this scheduler addresses the shortcomings of the CFS in terms of allocating time to multi-threaded programs, at the time of writing it is not integrated into the Linux Kernel, which still uses CFS by default. There are other scheduling mechanisms which have different goals in mind, for instance the scheduler in Windows, which is not fair in the case of a `REALTIME_PRIORITY_CLASS` process. A `REALTIME_PRIORITY_CLASS` process will always be immediately run, disregarding other processes [33].

As shown previously, improvements are possible. In 2016, 10 years after the introduction of CFS, Lozi et al. focused on the current implementation to find flaws and room for improvement in CFS [18]. The focus were four performance bugs which resulted in a percentage of the CPU cores being idle at times, while there were runnable threads in queues on busy cores. These bugs caused performance losses of 13-24%, with more extreme corner cases where performance losses were larger. These promising results are shared on GitHub and in the Linux Kernel Mailing List (LKML) [1] [2]. The issues are acknowledged by various 3rd parties and Linux Kernel maintainers. However, consensus is that the patches created as part of this research are causing other performance regressions, or are only beneficial for specific Linux platforms.

Moreover, additional work completed by Timm et al. [23] argues that operating systems would benefit from a central scheduler solution that leverages existing metrics to improve performance over a collection of threads. This is currently done by GCD (Grand central dispatch) used in iOS and OS X and offers "better efficiency than traditional threads" by handling thread creation and scheduling of asynchronous tasks on these threads at the system level [14].

Continuing on the topic of scheduling a collection of threads, thread pools can be improved as well. Chen et al. [2] have created a predictive model to forecast when maximum concurrency is achieved and contention cost is minimized. Besides that, a predictive model can also be used for determining the optimal size of a thread pool. This is done in work by Kim et al. [5], Kang et al. [6], and Lee et al. [13], where this prediction model is used to scale the thread pool, minimizing response times.

Previous research shows that coming up with scheduler improvements that are con-

[1] https://github.com/jplozi/wastedcores/issues/1
[2] https://lkml.org/lkml/2016/4/23/135

sistently better is difficult. What does become apparent is that there is performance left on the table, and that the scheduler can be improved.

2.2. BACKGROUND

To be able to quantify results, measure and investigate improvements in these areas, a layout of the basic system composition is required. In this section, composition and mechanisms of different parts of the Linux kernel are explained.

2.2.1. SCHEDULER

As previously stated, the OS runs on the CPU, making use of processes and threads. Once a program starts, a process is started. This process has one main thread. From the Linux kernel perspective, a thread is referred to as a process [3]. All threads are created equally, whether it is the main- or child-threads. In the kernel itself, the main thread has its thread group ID (TGID) set equal to its process ID (PID). A child thread, on the other hand, has its TGID set to its parent's PID. Other than that, a child and main-thread are equal in the kernel and are treated equally.

The scheduler has a difficult task to fulfill, the scheduler's job is to keep the processor busy with work, taking into account all the different threads that require running, while dividing processing time equally. Realization of these goals can vary per thread in the scheduler. Threads can be divided in two groups as described by Groves et al. in [9], these two groups have the following characteristics:

1. **CPU/IO Burst Cycles**: large actions of computation and I/O actions
2. **Filling the I/O Gap**: pausing thread execution to let I/O complete and resuming thread execution after I/O completion.

This division in actions is caused due to the fact that I/O is handled not in the program itself but by the driver. This driver has to wait for the actions to be performed on the respective device. Only once the device is done processing, the thread that was left waiting can continue normal execution. In the meantime, other threads are scheduled as to maximize the CPU usage.

The kernel documentation [26] shows that the CFS uses virtual runtime to tally the total consumed processing time of the CPU. The scheduler simply tries to keep this property of the thread equally distributed over all running threads. The CFS has one scheduler per CPU, which requires inter-core balancing of tasks which require processing. Once a scheduler for a core runs out of "jobs" to run, it checks whether there is another thread on another core waiting to be processed. This "work stealing" scheduler has to weigh the downside of moving the thread and possibly moving the accompanying data versus the gain of leaving the thread the original core and waiting for it to start there. CFS is straightforward and seems to be effective, since the later implemented BFS seems to underperform compared to CFS in terms of turnaround time. The improvement BFS

[3] https://linux.die.net/man/2/clone

introduces over CFS is in terms of latency [9]. Coming up with new schedulers is a tricky business where improvements are not guaranteed. Linux has been using the CFS since Linux 2.6.23, released in 2007 [4].

2.2.2. THREAD POOLS

These threads, which enable programs to perform work concurrently do have distinct downsides. For each thread to start, an amount of work is required: allocating memory and starting the thread itself. Upon completion of the thread, additional work is required cleaning up. These issues can be resolved resolved by utilizing a thread pool [1]. A thread pool has pre-existing threads, ready for execution. Once a job or task to complete arrives, it is assigned to an already running thread, skipping the work required to set it up. Thread pools offer performance gains to already concurrent applications, and partially remove of the overhead required for concurrent programs.

An example of an application which benefits from a thread pool is a backend server which is expected of handling a high amount of traffic. Without a thread pool there would be a significant penalty for spinning up a thread-per-request. With the thread pool it is easy to handle the request and return the thread to the thread pool, ready to be used for the next incoming request. Of course there are many other ways to leverage a thread pool, such as thread-per-connection or a way to schedule jobs to be run while reducing contention.

In most programming languages, there is the option to interact with OS-threads, there is the option to leverage a library, a package or the language itself. In these components, there often is a thread pool implementation, which provides a language-specific interface. This interface handles creation of OS-threads, leaving the programmer to fulfill more important tasks. If direct control of OS-threads is required, that's still possible, but these language specific thread pool implementations offer an easy to use interface. Examples of languages that offer this are Java (`ThreadPoolExecutor`), nodeJS (libUV), Rust (implemented in packages), Go (goroutines). If the existing implementation is not satisfactory, there is always the option to interface with OS-threads directly.

2.2.3. LINUX NETWORKING STACK

Improvements to the scheduling and related aspects can only go so far, as most applications have significant network involvement during execution. As shown by Uta et al. [20], applications with a network bottleneck are most impacted due to performance variability in the system. Network-bound applications have a slowdown up to 1.79, while CPU-bound applications have a maximum slowdown of 1.48. Removing these penalties incurred due to hardware variability will result in valuable improvements.

The Linux network stack has a number of functions that improve network related performance [31]. These mechanisms to remove these bottlenecks in the system are implemented in several places throughout the networking stack. Currently, five of these

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

mechanisms exist, aimed to improve network performance in multi-core systems.

Receive side scaling uses multiple queues. Incoming packets which are (likely) related will be grouped in the same queue, allowing processing of these packages to take place on the same core. The splitting is done by taking a hash of several packet headers and using portion of that hash to determine the queue. Related packages are now processed on the same core consistently.

Receive packet steering performs package grouping in software. RPS can be a bit redundant if there more buffers than cores, but the benefit is that there is no dependence on the NIC. In the case that there are more cores than buffers, RPS can split up the packets over all cores, instead of one core per buffer.

Receive flow steering is the next step in processing packets. RSS and RPS both divide packets over the available cores, providing an even load over the entire processor. RFS steers incoming packets to the core on which the packets are needed in userspace. The destination for a packet is determined with the use of the hash. The destination for the packet is found by using the hash and a lookup table to determine the final destination.

Accelerated receive flow steering is an optional component which is not supported by all drivers. A-RFS routes packages to the core where the process/thread is running, or to a core that shares cache with the core the process is running on.

Transmit packet steering tackles the situation where there are more queues for sending than cores. XPS divides the available queues over cores to lower contention. Cores are now not working in each others' transmission queues.

These methods rely on interrupts to schedule the correct threads once a packet arrives [31]. In the situation where there are more threads with network activity than queues for receiving and sending messages, other threads will be halted in favor of threads where an interrupt is issued. This could lead to (excessive) thread switching, or contention in the network.

All these different mechanisms try to improve the performance of the program in different steps in the networking chain. At the same time, the scheduler is starting, stopping and occasionally moving threads between cores. These two mechanisms have to work perfectly together to achieve the best performance. The resulting interaction between components is complex to analyze.

2.3. METRICS

To determine where we can improve programs using CPU or network metrics, we first have to correctly collect these metrics. For both the CPU and network usage measurements, there are multiple options. We will explore the available options and see what they have to offer. To get the best result with resolving contention issues for both the CPU and network scheduling, fine-grained and individual measurements are required. To see whether any inter-process or inter-thread bottlenecks are created, isolation of measurements on a thread-basis is best. Isolating the thread measurements allows spotting any specific issues with the scheduling, and resolving these issues effectively.

As a program executes, there is a likely situation that the program or its processes have different phases. Having thread specific information is critical to enable actions on a per-thread basis. For evaluation purposes, graphs of these program properties are required, not just final numbers of CPU and network usage, since inter-thread actions could have influence on each other.

2.3.1. NETWORK MEASUREMENTS

From the previously made statements in 2.3, potential measurement options can be selected. Looking for existing solutions where the required information can be directly extracted from the operating system is preferred.

SYSFS

The Linux filesystem has a variety of monitoring tools built into it. These tools are all accessible under the `/sys` directory [36]. These directories and files provide a way to get information about "devices, kernel modules, filesystems and other kernel components". These methods are used by Jannes Timm [22], and seem to work effectively for disk interfaces.

Network related data that is accessible from the filesystem is the data that originates from the various network interfaces available in the system (real and virtual). The first option is `/sys/class/net/eth0/statistics/{rx_bytes, tx_bytes}`, with `eth0` being the network interface and `rx` and `tx` distinguishing between receiving and sending respectively. The first limitation from this approach is that the information presented is not on a per-thread basis. Even worse is that the interface statistics don't distinguish between processes. These properties just keep track of the aggregate amount of bytes written and read in total for the interface, this makes it infeasible to evaluate multiple programs and isolating their data.

There seems to be an alternative in `/proc/${pid}/net/dev`, which should contain this information on a per-process basis, as the path suggests. Upon more careful investigation, this is definitely not the case. This location relays the information presented in the `/sys/class/net/eth0/statistics/{rx_bytes, tx_bytes}` location as mentioned above. This different location that appears to have per-process information shows per-interface information, accumulated from all processes. As per the requirements, this is not sufficient. This behavior from the Linux kernel is quite unex-

pected. The location suggests that the information is only the transfer information for the respective thread, however, the information is an interface aggregate placed in that location. Expectation would be that per-thread information is directly available from the kernel.

SYSTEMTAP

Systemtap seems to be a realistic option, which offers a way to execute code hooking into kernel functions with a low performance impact. Deploying systemtap tracing involves significant more work and can be described as quite error-prone. As explained in Brendan Gregg's blog [5], the simplified steps for using Systemtap are:

1. Add systemtap package, kernel headers
2. Get kernel debuginfo
3. Getting kernelversion the same as other packages
4. Patching systemtap C code and headers for build system
5. Power cycle on possible kernel freeze

These steps are involved, while the instructions also mention that the entire process is error-prone. Ease of use seems to be better in Red Hat Enterprise Linux, but relying on a single Linux distribution is not desired. More recent conversations about Systemtap are quite sparse and reactions in the past were not positive. It seems from recent information that Systemtap has improved greatly [6]. Systemtap is a possible solution, but the pitfalls in deploying it and the preference of Red Hat Enterprise Linux are downsides.

NETHOGS & PCAP

Nethogs is a user-space application which tracks network use on a per-process basis, but has an `htop` format. It uses the pcap library to measure the messages being sent on a per-process basis. The focus is heavily on current data and observing currently active programs, logging data is not a priority. To create logging with Nethogs, another program should read the data written to the terminal and interpret that data again, resulting in a suboptimal solution in terms of quality and efficiency. However, the underlying library used for Nethogs network measurements, pcap, is a viable option to measure network usage per process, but will require integrating that into the programs to observe.

PROXY

Proxying the connections is not a viable option, since there would only be process aggregates for the data transferred. Additionally, the proxy adds a new component to the system, adding to latency and system load, while offering the same results as other options.

[5] http://dtrace.org/blogs/brendan/2011/10/15/using-systemtap/
[6] https://blog.openresty.com/en/dynamic-tracing-part-2/#systemtap

TASKSTATS

The taskstats interface is described in the kernel documentation as "Taskstats is a netlink-based interface for sending per-task and per-process statistics from the kernel to userspace." [29]. In essence, each Linux process keeps track of resources which are implemented in the taskstats system. An important remark relevant for taskstats is that from a kernel perspective, a thread is equal to a process. This means that taskstats is able to keep track of the data on a per-thread basis. All the fields which are or can be tracked can be seen in the `taskstats.h` file in the Linux kernel code [27]. Examples of these fields are: process exitcode, process/group related id's, user/system cputime and read/write I/O bytes.

To communicate data to userspace, a netlink-based interface is used. Netlink is a "socket-based interface"[7]. From a process, you can establish a connection with the Netlink interface and request the desired data by sending specific taskstats messages, or subscribing to receive messages issued by taskstats.

Contrary to pcap, which intercepts all packets on the system, taskstats works on a per-thread or per-group basis. In addition to being able to request the statistics to be sent, subscribing to exit data from exiting processes is also an option. A userspace process can register to receive exit data from specific CPU cores by using a `cpumask`. This `cpumask` signifies the cores from which exiting tasks' information should be sent to the userspace process. For instance: `"1,2,3,10-12"` would result in taskstats sending the exiting threaddata from tasks running on 6 cores, of which 3 are in a range of cores. The data sent on thread-exit is the same data as the data that can be requested during execution, just sent after the execution is done and final numbers are in.

The main issue with the taskstats interface is that it does not include network statistics. This will have to be changed for taskstats to be a suitable solution. The taskstats interface relies on its internal `taskstats`-struct [8], which is a struct linked to each thread in the kernel. This `taskstats`-struct is modifiable to include network data on a per-process basis, allowing that data to be transferred by taskstats as well.

2.3.2. CPU MEASUREMENTS

Obtaining CPU measurements is more straightforward in the Linux kernel. The previous networking section shows that the taskstats interface has the most potential to provide continuous and per-thread measurements. Previously mentioned, the taskstats accounting data includes the fields for user- and system-cputime. Other options would be using systemtap, which would have the same cons as for the usage of systemtap for the network.

2.3.3. STATE OF LINUX MONITORING

The current state of process or thread monitoring in Linux currently lacks tools to observe thread activity. Most tools seem to have a focus on ad-hoc monitoring, showing only current data. The best example of this is `htop` and other `*top` equivalent programs.

[7] https://man7.org/linux/man-pages/man7/netlink.7.html
[8] https://github.com/torvalds/linux/blob/master/include/uapi/linux/taskstats.h

Other solutions are very low-level, they cannot be used immediately and require manual configuration and setting up. The low level solutions (Systemtap, kprobes) they allow for far more usages than CPU and network graphing/measuring. However, to utilize these, the user is required to write code to get them to work, and do not produce an observable interface for the user. These lower level solutions are geared more towards low-level system engineers or administrators with debugging needs during development.

Due to this difference in goals, there exists a gap between the monitoring tools. On one side, the tools are too professional, aiming to allow for precise kernel debugging for system calls, kernel systems and driver. On the other hand, they serve a real user-usecase, exposing a basic type of monitoring which is quite simple.

2.4. REQUIREMENTS

If we want to have the monitoring system fill in the gap left by other Linux monitoring tools, we can set up the requirements for that system. The components below can provide all the metrics which allow further experimentation.

- Record per-thread information during program execution
- Graphs of CPU usage, network transfer and network system calls
- Record per-thread totals on program termination

From these requirements, combined with the analysis above, it seems that using the taskstats interface is best for realizing this monitoring system.

3

WORKLOADS

Supported by the background knowledge from Section 2, we can look into ways to validate and test the improvements we want to make. For this, we need ways to check the behavior of the adapted scheduler versus the default scheduler. Most importantly, we have to pinpoint a scheduler to target, since there are various options.

3.1. BENCHMARKS

To confirm whether we can make improvements to the performance based on the network or CPU statistics, we need to establish a meaningful baseline and select representative workloads which help us to determine the potential for optimization through scheduler changes. These benchmarks need to be operating in a concurrent manner while also producing sufficient load on the system.

3.1.1. LOW-LEVEL BENCHMARKS

Linux has a lot of benchmarks that test the system components, these benchmarks consist of synthetic workloads. This category of benchmarks is large for Linux, which is in line with the strong presence of low-level measurement tools mentioned in 2.3.3. Examples of these are Ethr[1], Uperf, iPerf and neper[2]. The issue with these benchmarks is that they serve a different purpose. These benchmarks attempt to test and validate the deployed network. The network is tested as an isolate, not as an entire system. Because the lower-level benchmarks do not take into account the entire system, we prefer to use real-world benchmarks which simulate or behave like a real application.

3.1.2. REAL-WORLD BENCHMARK

Finding benchmarks that satisfy these conditions seems to be difficult. A real-world workload is preferable, since execution, validation and improvements can be realized

[1] https://github.com/microsoft/ethr
[2] https://github.com/google/neper

and observed directly. A workload with these properties is a database workload, combining network-I/O with a CPU load. On the other hand, a web-API also satisfies these conditions. As described in the background 2.2.2, APIs often have a form of thread-per-request, which make them suitable workloads. For the workloads, a database and a web-server will be used.

3.1.3. LANGUAGE CHOICE

To be able to integrate our improvements into one component, without the need to build additional components, we want to have one programming language which will be used for collecting data, running and adapting the benchmarks, and scaling the thread pool. Implementing the taskstats monitoring system will be done using the same programming language as the benchmarks and for scaling the thread pool. Since the taskstats monitoring is at the basis of the solution, we will check whether there are any suitable taskstats implementations.

Initial exploration shows that there are readily made packages to interact with the taskstats interface. Looking at the available libraries to use as a starting point for monitoring taskstats data, it seems that one of the Go libraries [3] has the most taskstats features implemented and documented. Alternatives written in Python and Rust have little documentation and seem less complete. The Go library uses the genetlink [4] and netlink [5] packages for communication with the taskstats interface. All these packages provide ample documentation to start with. From this information, it seems that using Go for the implementation of solutions is a safe choice. Now that the Go language is chosen, we will also select benchmarks written in Go. This allows us to integrate the solution into the benchmarks themselves as mentioned above.

3.1.4. DATABASE BENCHMARK

As a suitable database benchmark, which we proposed in section 3.1.2, CockroachDB can be used. CockroachDB is a distributed database, which offers consistency in large-scale deployments [24], while also delivering the same experience as using a traditional single-node database. All the data is split over smaller ranges, allowing for these to be maintained, distributed and updated by a Raft consensus mechanism [16]. Once a range grows larger than a predefined threshold value, this range is split and re-distributed over the other nodes. This marks the duality of this workload, having to process SQL queries on one hand, while also making sure that replicas or ranges are distributed in the cluster. This should give a representative workload to test the system in a meaningful way. Another benefit of CockroachDB is the included benchmarking suite. This is a selection of a couple workloads from the included benchmarking suite [25]: a simple key-value benchmark, a bank workload which has currency balances and credit updates, the TPC-C workload [6] and the YCSB workload [10].

[3] https://pkg.go.dev/github.com/mdlayher/taskstats
[4] https://pkg.go.dev/github.com/mdlayher/genetlink
[5] https://pkg.go.dev/github.com/mdlayher/netlink
[6] https://www.tpc.org/tpcc/

3.1.5. THREAD-PER-REQUEST BENCHMARK

The second benchmark proposed in section 3.1.2, the thread-per-request workload, can be a highly concurrent workload. A big portion of the web is built following this architecture. Java has Apache Tomcat (and by proxy Spring [7]), NodeJS has libuv [8] and Go has its own scheduler combined with goroutines. For the benchmark, we'll use go-web-framework-benchmark (shortened to gwfb for brevity) [9]. This benchmark uses wrk [10] to generate a load, while gwfb tests multiple implementations of go web frameworks. This benchmark makes it possible to see how universal the results are in Go libraries, since the benchmark tests a variety of thread-per-request implementations. This benchmark is highly concurrent, the example from the repository is run on 8 1.8GHz cores and achieves around 50k requests per second, saturating the CPU. The implementation of the endpoints is simple. It has a `GET:"/hello"` route which responds with plaintext `"hello world"`. This benchmark also allows for each call to perform "work" while the endpoint handler is called. In this case, the work is not work, but a no-op in the form of a `sleep` call. This may not seem relevant, but this simulates making a call to a database or another http endpoint. If a measurement has the `_2ms` suffix, there is a 2ms sleep performed each time the endpoint is called. This sleep means that the Go scheduler has to handle more requests concurrently, requests are longer in the system and are replied to after more requests have arrived during the sleep.

3.2. BENCHMARK INTERNALS

To get a better idea of where we can collect the required network data that is not present in the process' `taskstats`-struct yet, we need to look into what the benchmarks are doing. More specifically for adding the network data to taskstats, we need to figure out how Go communicates with the network stack. Since taskstats works on the kernel level, we have to find the place where this data can be intercepted in the kernel. To find the place in the kernel where this information is relayed, we can use strace.

Strace [11] traces system calls a program makes during execution. Each time a system call is made, strace logs the arguments and the return value. To get a better overview, we can make use of the `--summary-only` flag to generate a report after the benchmark exits. This report contains a list of all the performed system calls, allowing a better look into the inner workings of Go. Table 3.1 is the result of running strace during a CockroachDB benchmark. Note that these system calls are a subset of all system calls, only network related system calls are selected here. The full table can be found in appendix A.

As shown in table 3.1, the network-related system calls which are responsible for sending and receiving data are: `sendmmsg, recvfrom, sendto, recvmsg`. If we want to measure network usage, we'll need to track the contents written and read by these methods and account for them in the taskstats interface.

[7] https://spring.io/
[8] https://github.com/libuv/libuv
[9] https://github.com/smallnest/go-web-framework-benchmark
[10] https://github.com/wg/wrk
[11] https://man7.org/linux/man-pages/man1/strace.1.html

3

time	seconds	usecs/call	calls	errors	syscall
0,00	0,000019	1	12		socket
0,00	0,000010	2	5		sendmmsg
0,00	0,000007	0	10		recvfrom
0,00	0,000000	0	1		sendto
0,00	0,000000	0	3		recvmsg
0,00	0,000000	0	4		getsockname
0,00	0,000000	0	1		getpeername
...	...	...	...		...
100,00	52,376266	12	4358322	74718	total

Table 3.1: strace results from CockroachDB instance, non-network calls removed from table

4

EXPLORATION

4.1. BENCHMARKING SETUP

For CockroachDB each benchmark is run `n` times, where `n` is the amount of runs required to achieve a standard deviation smaller than 5% over all runs. For each benchmark to run, the average and standard deviation are calculated. The gwfb contains multiple different frameworks to benchmark, these are all resulting in the same parameters being run multiple times, just with a different web framework. For gwfb, each parameter is run `m` times. `m` is the total amount of frameworks that are tested. The list of available benchmarks in gwfb is large (49 testable libraries), so to limit the running time of all experiments, a random selection of benchmarks to execute is made. In listing 4.1 are the chosen packages for testing, all available options are listed in appendix B.

Listing 4.1: selected benchmarks in go-web-framework-benchmark

```
web_frameworks=("gorilla" "fasthttp" "gin" "httprouter"
                "fastrouter" "martini" "clevergo")
```

The benchmarks are run on a system with an AMD 3700x with 8 cores clocked at 3.6GHz, 32GB of memory and an SSD. The OS is a clean install of Ubuntu 21.04. Additionally, for final result validation benchmarks are run on nodes in the DAS6 cluster [17]. These nodes allow the benchmarks to run on a dual 16-core AMD EPYC-2 (Rome) 7282 at 2.8GHz and at least 128GB memory, while also being able to read power consumption using `likwid-powermeter` [1].

An important detail of the two types of systems that the benchmarks are executed on is that the consumer hardware executes everything locally. This means that there is no actual network involved, the load generator and the programs to benchmark are running on the same machine. On the contrary, with the DAS6 cluster, the programs to benchmark are executed on one machine, while the load generators are running on another machine. Communication is done over the network.

[1] https://rrze-hpc.github.io/likwid/Doxygen/likwid-powermeter.html

4.2. MONITORING

To build the entire monitoring system outlined in 2.3.1, we need to make changes to the programs that will be tested. In section 2.4 the requirements for this system are listed. Components that are needed to fulfill those requirements are:

- Extend taskstats interface
- Request data from taskstats
- Listen to exit data from taskstats
- Creating metrics, graphs and connecting components

4.2.1. EXTENDING TASKSTATS

Taskstats currently only has file read/write statistics in the form of `read_bytes` and `write_bytes`. Next to these, there are the `read_syscalls` and `write_syscalls`. These fields are similar to what can be used for the network measurements, so we can extend taskstats with `read_net_bytes`, `write_net_bytes`, `read_net_syscalls` and `write_net_syscalls`.

Listing 4.2: Taskstats fields for tracking network activity

```
struct taskstats {
    ...

    __u64   read_net_bytes;      /* bytes of network read I/O */
    __u64   write_net_bytes;     /* bytes of network write I/O */

    __u64   read_net_syscalls;   /* read network syscalls */
    __u64   write_net_syscalls;  /* write network syscalls */
}
```

4.2.2. POPULATING DATA

To get the relevant data into the taskstats interface, changes have to be made to the Linux networking internals. The methods found in 3.2, `sendmmsg`, `recvfrom`, `sendto`, and `recvmsg`, are extended to add their data to the taskstats struct for each process. With all system calls now adding data to the taskstats struct, the benchmarks will have their network usage tracked.

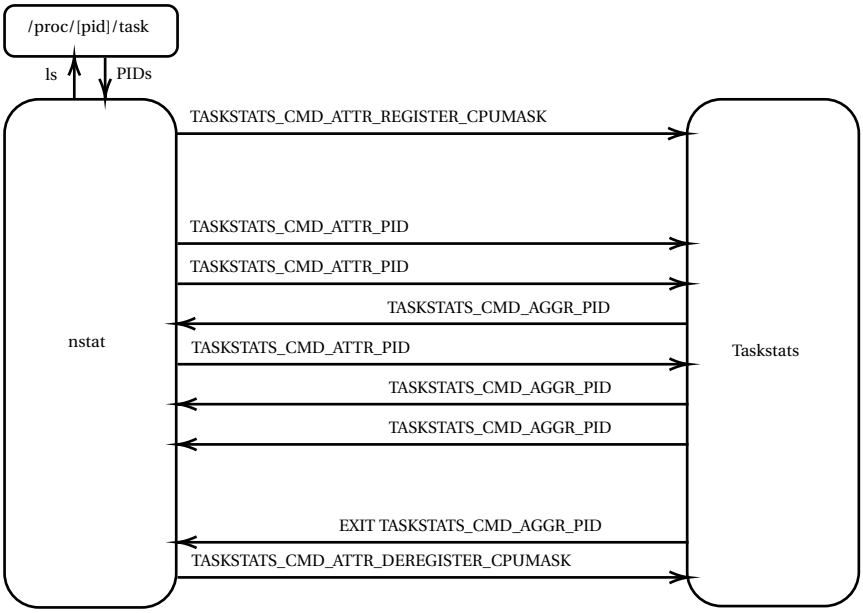
4.2.3. GETTING THE TASKSTATS DATA

Retrieving the data from the taskstats struct involves sending messages over the taskstats interface described in section 2.3.1. The message `TASKSTATS_CMD_ATTR_PID` with as body the respective PID of the process results in a reply with the data that taskstats has for that PID. In the case where for instance the program is running with a PID of 41, but the program has 2 threads, taskstats will still only reply with the requested data for the main thread with PID 41. To get the taskstats data for the other thread 42, an

`ls` command of the filesystem location `/proc/41/task` is needed. This `ls` command will reply with `[41, 42]`, the PIDs of sub-processes of the main thread, including the PID of the main thread. To get all data belonging to all individual threads, sending a `TASKSTATS_CMD_ATTR_PID` for each of the returned PIDs from `/proc/[pid]/task` is required. An illustration of this can be found in Figure 4.1. The monitoring solution that is the result of all these components is `nstat`, as mentioned in the figure.

The complete flow of messages handled by nstat is outlined in Figure 4.1, it looks simple, but there are some intricacies. With handling taskstats messages, responses for all commands are received on the same socket connection. There is no guarantee that after sending a `TASKSTATS_CMD_ATTR_PID` message, the immediate response is for the message sent. The response can also be a message containing the summary of an exited thread. The incoming messages have to be filtered on the headers to figure out what data is contained and what triggered the message.

Figure 4.1: Design for nstat



Taskstats also has the option to listen to exit data, which is sent once a thread or process exits. In such a message, the cumulative data from that thread is sent to the listener in userspace, where the totals for that thread can be processed. After observation of all threads is done, it is advised to deregister the listener for exiting processes, otherwise connections are only cleaned up after the kernel decides there are too many open connections [29]. Not cleaning up the registered listener a couple of times does introduce penalties in message speed for exit data.

time	seconds	usecs/call	calls	errors	syscall
1,15	0,603630	3	167650		write
0,31	0,161232	2	58415	28521	read
0,00	0,000019	1	12		socket
0,00	0,000010	2	5		sendmmsg
0,00	0,000007	0	10		recvfrom
0,00	0,000000	0	1		sendto
0,00	0,000000	0	3		recvmsg
0,00	0,000000	0	4		getsockname
0,00	0,000000	0	1		getpeername
...	...	...	...		...
100,00	52,376266	12	4358322	74718	total

Table 4.1: strace results from CockroachDB instance, including write/read calls used over the network

Now that network and CPU usage graphs of programs are available, we can start exploring whether the results are accurate. For that, we look at a small (custom) program called `get` that retrieves a webpage a 100 times. It downloads the webpage passed as an argument and sleeps for a second before performing the next download. Upon running this program and evaluating the results, the amount of data transferred over the network for one request can be checked with curl:

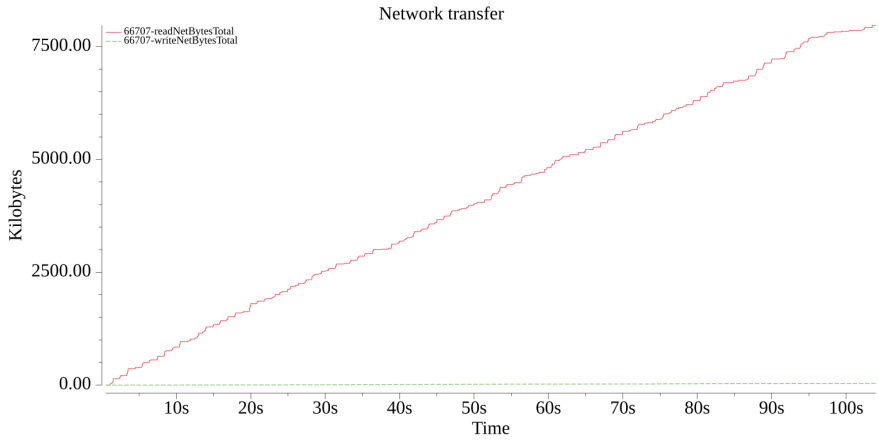
```
curl -compressed -so /dev/null https://nos.nl -w '%{size_download}'
```

The result obtained from `curl` has a size of 79KB. Comparing that to the amount of data transferred over the network by `get`, across its 100 iterations, get registers less than 79KB. This shows that the data collected from the system calls is incomplete. After more thorough investigation, the explanation can be found in the kernel documentation. In the kernel documentation for sockets [34], it is mentioned that a normal `read` or `write` system call can be used to write to the socket. That could explain the big difference in results. If we now refer back to the results for strace run on CockroachDB in Table 4.1 (full version in A), we can see that there is a substantial amount of `read` and `write` system calls that could contain a big amount of writing to a socket. Upon further investigation, it seems that these methods detect whether they are writing to a socket. If the system call is indeed writing to a socket, the correct methods in the networking code are called for sending or reading the data over the network.

To fix this issue, we incorporate the methods responsible for writing and reading to and from the network using the filesystem syscalls. The `read` and `write` methods utilizing networking code when the destination for the calls is the network are handled the same as normal network calls now. Running `nstat` and the `get` program again returns 7895KB of data transferred, the accompanying graph for the network transfer can be seen in Figure 4.2. In this figure the red line indicates the bytes read from the network over the entire duration of execution. The green line shows bytes sent, which are close to zero, since this is just a `GET` request. The bytes read shows the stepwise increases due

to the second idle between each retrieval of the page. Confirming whether the results are accurate by checking with `curl`, using the same command previously mentioned, reports back that the amount transferred is 79326B. Again multiplying with the amount of times the page is retrieved yields a total of 7932KB that should be measured by `nstat`. Transfers for this page have fluctuations of 7B between them, so the results from `nstat` are now accurate, considering that 7932KB was expected and 7895KB was measured.

Figure 4.2: Total network transfer of `get` program, measured by `nstat`, showing in red the read bytes and in green the sent bytes during execution



4.3. SOLUTION EXPLORATION

To explore several options for improving contention from a network or CPU perspective, we're setting up a 3-node CockroachDB cluster. Each node is connected to two other nodes, creating complete graph network. After benchmarking with the bank workload over 1 minute, the transferred data is evenly distributed between nodes and totals 660MB. The average CPU usage for these nodes adds up to 13.4% in total for all nodes. For a TPC-C benchmark, 840MB was transferred between instances over a period of 13 minutes, with an average CPU usage of 12%. This shows that the TPC-C workload is not suitable for generating enough workload for the network, since over a span of 13 minutes, it only results in generating a 1MB/s network load over all three nodes. The bank workload is capable of generating a network load of 11MB/s for all nodes. This comes down to about 4MB/s per node, which is more than the TPC-C benchmark, but not a quantity that suggests that a bottleneck is reached. This is further investigated in the following section.

4.3.1. NETWORK

Following this initial step, we're introducing several changes to CockroachDB to see whether changes to the network have any impact on the performance. Introducing additional latency for messages between database instances has no influence on performance, it only results in queries no longer being distributed and replication being halted. Adding

latency to the gossip mechanism used to relay data between instances only resulted in the load being more spread-out over a longer time. Range distribution over the other nodes for replication takes around 40% longer in that case. It seems that influencing the network or network-related actions do not seem to achieve anything in terms of performance. With only 4MB/s of network traffic per node, the amount of traffic is too low to cause a bottleneck on the system for the distributed 3-node CockroachDB cluster.

4.3.2. CPU

Following the results related to network traffic, we now look towards CPU usage. Go is a language with an internal scheduler. All Go code runs in a so-called goroutine, which is a thread handled by the Go scheduler. Under the hood, these goroutines run on the OS threads which are created and taken care of by the go scheduler. With the `GOMAXPROCS` setting, the amount of threads that are allowed to progress concurrently can be changed. In the default case, an amount equal to the processor threads are allowed to progress at the same time. When changing the value of `GOMAXPROCS`, the amount of threads progressing changes immediately. This allows for a great way to influence how the Go scheduler behaves, it responds immediately to changes.

For instance, on a CPU with 4 cores (8 threads), by default there would be 8 OS threads running for a Go program (**A**). Once you start a second Go program (**B**), another scheduler is started to run that program. The scheduler for **B** also uses 8 OS threads, while disregarding the fact that there already was a go scheduler running for **A** which uses 8 OS threads. These schedulers are now competing with each other for the same compute time in the CPU threads. In total there are 16 threads started, even though there are only 8 CPU threads available. This doesn't have to be an issue though, since this is something the OS scheduler can and should take care of. It could be more logical to provision 4 OS threads for each go scheduler in this situation, for a total of 8 OS threads. This could reduce the times that the OS scheduler changes the currently active thread from **A** to **B** and vice versa.

4.4. VALIDATION

To validate whether we can use an algorithm to steer the value for `GOMAXPROCS`, we perform a couple of runs for validation. While exploring potential improvements, there seemed to be a peak in total operations just below the maximum value for `GOMAXPROCS`. To verify this, 2 separate 1-node instances of CockroachDB are used, without any form of clustering. It will be run on the 8-core system described in section 4.1. The workload to be used is the bank workload, since this workload does not use transactions. While doing other tests like TPC-C, it seems that the transactions in other workloads slow down processing so much that the difference is negligible. With n runs of 3 minutes per `GOMAXPROCS` setting, stopping when the standard deviation is approximately 5%, the average is calculated over all runs.

Figure 4.3 shows the results, on the x-axis the values for `GOMAXPROCS`, on the y-axis

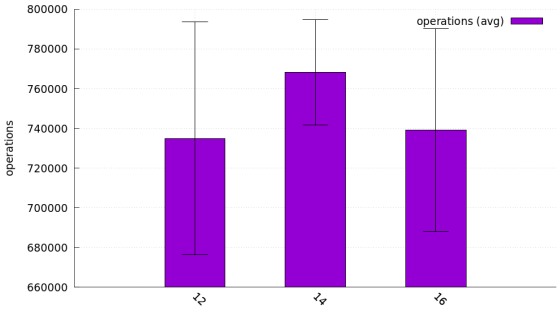


Figure 4.3: Total operations per GOMAXPROCS of two instances of CockroachDB running bank benchmark

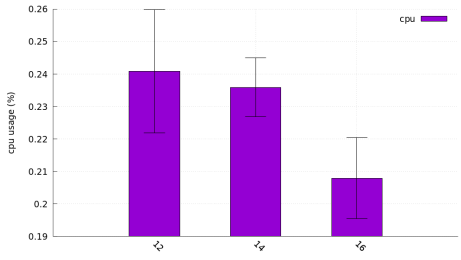


Figure 4.4: CPU usage vs GOMAXPROCS, CPU usage relative to the entire CPU

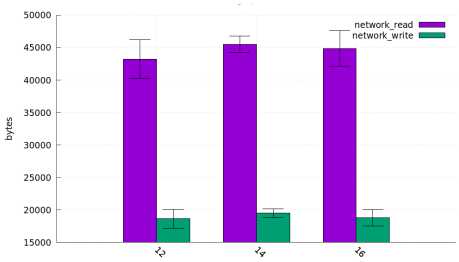


Figure 4.5: Network usage vs GOMAXPROCS

the operations. The operations metric is constructed by keeping count of all completed actions performed over the database connection by CockroachDB. We can see that performance improves when scaling GOMAXPROCS down when running 2 non-clustered instances. The improvement in operations is 3.9%. The standard deviation is also lower for the GOMAXPROCS=14 case, results are more consistent. The runs for GOMAXPROCS=12 and 16 have runs with significantly higher or lower total operations, causing the increased standard deviation. These outliers are not consistent and appear to be random.

Figures 4.3, 4.4 and 4.5 show the results from the exploratory runs of CockroachDB to determine the most effective metric for steering the GOMAXPROCS variable. We can see from the different values for GOMAXPROCS that the CPU usage varies between them. From Figure 4.3 and 4.4, we can conclude that there is no relation between the CPU usage and the operations performed. The highest amount of operations at GOMAXPROCS=14 is not a result of the highest CPU usage, since that is recorded at GOMAXPROCS=12.

With Figure 4.3 and 4.5, we can see that the GOMAXPROCS value for most operations is the same for most network traffic measured at GOMAXPROCS=14. The network usage from the default GOMAXPROCS=16 to the optimal GOMAXPROCS=14 causes an 1.3% increase in network usage. Comparing this to the difference operations performed for GOMAXPROCS 14 and 16, which had a difference of 3.9%, the operations performed had a larger percentual improvement. From this we can conclude that steering on operations is a better

option that steering on the network.

5

UNIVERSAL SOLUTION

Now having found a way to improve the performance of CockroachDB by using the `GOMAXPROCS` variable, we can create automated solutions to achieve our goal. Simply setting `GOMAXPROCS` to a value is not going to be enough for all use-cases. Additionally, selecting a percentage of cores to progress, for instance 80%, will not take into account the changing nature of deployments and different kinds of programs. A program with internal messaging might run into issues when there are eight internal message channels, while running with `GOMAXPROCS=6`, causing a bottleneck. The goal must be to set `GOMAXPROCS` taking into account different types of programs, with no help from the user when selecting this value.

To achieve this goal, we want to use a solution that is able to find the best value for the program during execution. Being able to find the best solution is nice, but not every program consistently behaves the same way. Some programs go through several different phases of execution, or have background tasks that are started during execution. To handle these changing scenarios without interrupting the program, it should be possible to rebalance the `GOMAXPROCS` value in a way to still provide the best results when this happens.

For both solutions, there is shared code required to allow the solutions to interface with the programs to improve. To keep track of how much operations are performed, there is an `ops` variable that is exposed globally. This variable can be atomically increased to relay to the algorithms created how much operations are performed since the previous interval. In this case, an operation can be defined in any way you want. For instance, execution of a SQL query could be a reason to increment `ops` by 1, serving a HTTP endpoint or sending a message on a channel could be another reason. This `ops` variable will be atomically read in the solution algorithms.

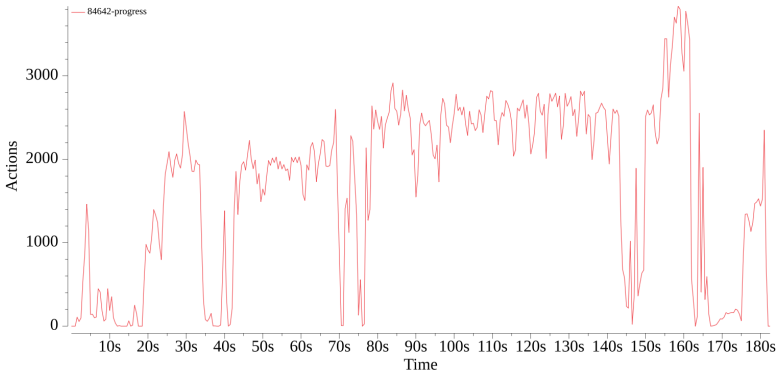


Figure 5.1: Inconsistent progress in CockroachDB

5.1. ALGORITHMS

For our solution to be able to find the best value for `GOMAXPROCS`, we start of by implementing a hillclimbing algorithm. This algorithm is an initial simplified version of the one presented by Jannes Timm [22]. Unfortunately, due to the programs to benchmark, this solution proved to be unsuccessful due to the nature of CockroachDB's internals. As an elegantly simple alternative, we propose linear search, which proved to work according to our initially performed experiments in section 4.4. There seems to be one best value for `GOMAXPROCS`, and not multiple. The value that will steer the algorithm is the operations performed during a time interval. For CockroachDB, this is the amount of commands that are handled over the database connection. For gwfb this is the amount of requests handled. As noted in section 4.4, the operations value had the biggest percentual difference between measured values for `GOMAXPROCS`. Compared to the other options like CPU usage and the network transfer, steering directly on operations is the best choice for the algorithms.

After implementing and testing the initial linear search algorithm, we will evaluate it's performance and issues that come to light. Based on these observations, an improvement is proposed which should alleviate the observed issues.

5.1.1. HILLCLIMBING

The algorithms used in the research performed by Jannes [22] is implemented to create a baseline measurement. This hillclimbing solution starts at the minimal value for the thread pool. From there it explores upwards until a peak in performance is found and settles on that value. After a timeout, it explores downwards or upwards again under the condition that performance is improved significantly. This solution has shown effective for the applications selected in that respective research. However, after running that solution with the CockroachDB benchmarks, it has shown to be ineffective. Due to large differences in CockroachDB performance likely caused by background tasks, the solution always performed significantly worse than the default.

This behavior can be seen in Figure 5.1, where there are several large dips where performance is close to zero. Each time these dips in performance occur, the hillclimbing algorithm makes an incorrect decision which results in a longer period of non-optimal behavior. The hillclimbing method then takes a while to recover, after which it quickly encounters another such a low reading. This process results in the algorithm continuously making wrongly informed decisions.

5.1.2. LINEAR SEARCH

As the first solution was not successful, we propose a new algorithm for finding the best value for `GOMAXPROCS`. Linear search searches the entire search-space once and selects the best performing value for `GOMAXPROCS`. This method uses a global file to coordinate between instances. Upon startup and shutdown, program identifying data is written to the shared file, this to ensure that the linear search is not continuously restarted. Each time changes are observed to the file, the linear search is restarted and runs until it reaches the `PAUSED` state. Additionally, programs are able to manually restart the search phase, for instance, when the program reaches a different phase in execution. In the case where a program needs a long setup time which is single threaded, and it later switches to using all threads, it can signal that a new search phase is needed to make sure that it is using the optimal value for `GOMAXPROCS`.

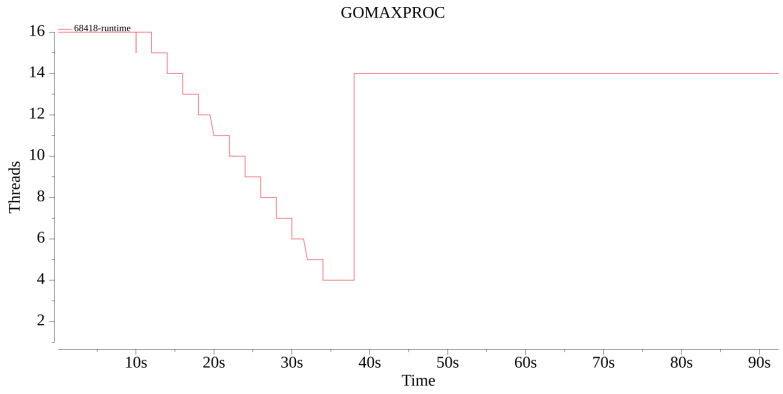


Figure 5.2: Behavior of Linear search algorithm setting `GOMAXPROCS`

The algorithm can be seen in Algorithm 1, accompanied with its behavior for selecting `GOMAXPROCS` in Figure 5.2. The actual implementation of linear search is slightly different from the one shown in the algorithm, since there are issues that are specific to the CockroachDB benchmark. The CockroachDB benchmark needs time ramping up before a stable throughput is reached, so the linear search waits 10s before starting the search phase. Additionally, there is a restriction imposed that ensures that the algorithm cannot go under values that are too low (`GOMAXPROCS=4` in this case). This constraint avoids testing very low values at which point performance is limited, even when there are no competing thread pools on the system.

Algorithm 1 Linear search

```
cores ← amount cores

ops ← 0
prev_ops ← 0
progress ← 0
state ← SEARCH
buckets ← {}

for every 2 seconds do
    prev_ops ← ops
    ops ← load operations value
    progress ← ops − prev_ops

    if should restart then
        cores ← amount cores
        ops ← 0
        state ← SEARCH
        buckets ← {}
        setGOMAXPROCS(cores)
        continue
    end if

    if state == SEARCH then
        buckets[cores] ← progress
        if cores > 1 then
            cores ← cores − 1
            setGOMAXPROCS(cores)
        else
            state ← FOUND
        end if
    else if state == FOUND then
        max ← key of max value from buckets
        setGOMAXPROCS(max)
        state ← PAUSED
    else if state == PAUSED then
    end if
end for
```

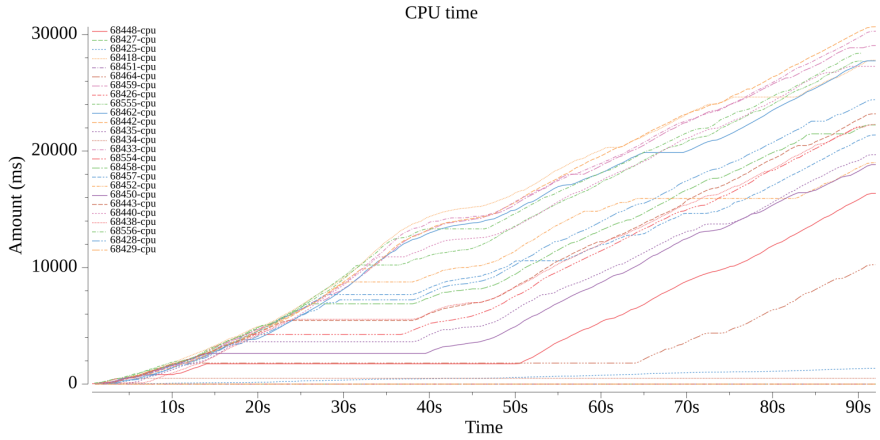


Figure 5.3: `nstat` showing per-thread behavior of CPU threads while performing linear search, with increasing slope due to Go allocating more CPU time due to a decreasing amount of threads to schedule from 10s-38s

5

If we look at the two figures 5.2 and 5.3, the relation between the value of `GOMAXPROCS` and the CPU usage per thread can be observed. Once the algorithm starts lowering the `GOMAXPROCS` value, the Go scheduler is stopping progress for threads. During the search, the amount of threads using CPU is decreasing, allowing other threads to pick up the slack and start using more CPU time. This is seen in the graph by the increasing slope of the lines. With less threads competing for the same resources, there is a potential to improve performance due to reduced contention between the threads. At the 25s mark, the linear search method schedules more threads after increasing the `GOMAXPROCS` setting, this reduces the slope of the progressing threads again. This example really shows to what extent the individual thread monitoring solution gives insight into what is happening with individual threads on the system.

5.1.3. BIDIRECTIONAL HILLCLIMBING

The linear search method previously described can be improved upon, mainly the fact that the search phase tests non-optimal values. While testing these non-optimal values during the search phase, the lower operations performed lowers the total throughput of the system. In cases where the search phase has to take place often, this results in poor results. To alleviate this, we look towards an improvement that can be best described as bidirectional hillclimbing. In Algorithm 2, the algorithm continuously searches for the optimal value. `GOMAXPROCS` starts out equal to the Go default value, which is the amount of threads the CPU has. It then travels over the search space, adjusting the value of `GOMAXPROCS` accordingly, while checking each step whether previously made progress was greater than the current progress. Once the previously made progress is more than the current progress, it goes back to the previous value for `GOMAXPROCS` and reverses its search direction. If the bounds of the search space are reached, the direction is also reversed. The long climbing time for the algorithm described in Section 1 is no longer an issue for this algorithm, since it reverses immediately when performance is decreasing.

This solution is also suitable for cases where the workload changes during execution, due to the fact that the search phase has no termination. This should remove the burden on the program to handle these phase changes during execution.

Algorithm 2 Bidirectional hillclimbing

```
max_cores ← amount cores
cores ← max_cores
prev_cores ← max_cores

ops ← 0
prev_ops ← 0
progress ← 0
prev_progress ← 0

step ← 1

for every 2 seconds do
    prev_ops ← ops
    prev_progress ← progress
    ops ← load operations value
    progress ← ops − prev_ops

    if prev_progress > progress then
        setGOMAXPROCS(prev_cores)
        step ← −1 ∗ step
        continue
    end if

    if cores == max_cores then
        step ← −1
    else if cores == 1 then
        step ← 1
    end if

    prev_cores ← cores
    cores ← cores + step
    setGOMAXPROCS(cores)
end for
```

In Figure 5.4, the searching behavior of the algorithm is shown. The algorithms starts at the default value, `GOMAXPROCS=16`, the amount of cores on that machine. Directly it starts searching in a downward direction. Once progress no longer increases, the direction reverses. From the graph, it seems that the optimal value for `GOMAXPROCS` is somewhere around 10-15 according to the bidirectional hillclimbing algorithm.

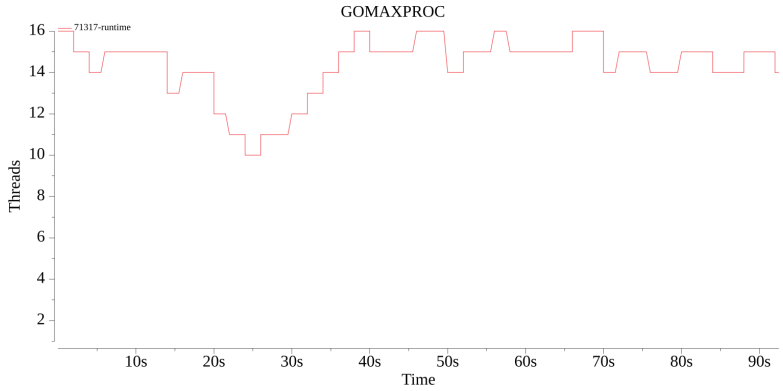


Figure 5.4: Behavior of bidirectional hillclimbing algorithm setting `GOMAXPROCS`

5.1.4. PECULIARITIES

The algorithm can temporarily reside in a state of alternating between two values, while keeping one of these values could result in greater performance. The fact that the search phase has no end condition coincides with the constant switching between two values. In cases where the load does not change, the value for `GOMAXPROCS` does keep changing. Once the best value is found, it keeps selecting a value one higher or lower than that value and then going back to the optimal. To make this algorithm work better with loads that do not change, exponential back-off can be implemented in a way where the search phase is paused and started again after an increasing interval.

In addition to that downside, the observation previously made that for CockroachDB, the total operations performed during the two-seconds measurement window can fluctuate heavily. The hillclimbing algorithm did not perform at all, but this also affects the two newly proposed solutions. For the linear search and bidirectional hillclimbing methods, this occasionally results in the selected value for `GOMAXPROCS` not being optimal. CockroachDB has underlying processes which hinder performance during intervals. For the linear search method, this only matters if it happens during the search-phase. In the case of that happening, the value is settled on is not optimal. If this happens in the bidirectional algorithm, the algorithm deviates from the optimal value. Recovering from this deviation to the optimal takes time, this can be observed in Figure 5.5. With the bidirectional algorithm, it is less of an issue, because it quickly returns to a value close to the optimum.

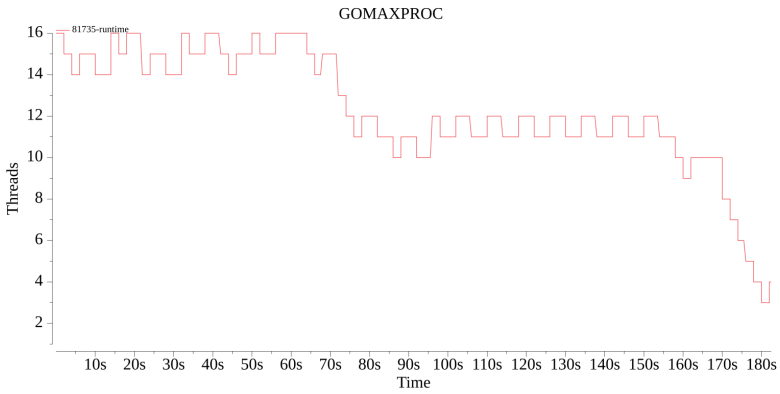


Figure 5.5: Drifting of bidirectional hillclimbing algorithm

6

EXPERIMENTAL RESULTS

For evaluating the performance of our improvements and to validate that the solution applies to different types of hardware, we make a distinction between our two types of hardware, consumer and server hardware. Benchmarks are run on an 8 core (AMD Ryzen 7 3700x) and a dual 16-core system (2x AMD EPYC-2 Rome 7282), the full details can be found in section 4.1. The 8-core consumer system has less cores, while being clocked at a higher speed. The dual 16-core server system has a lower clock speed, but four times the total amount of cores.

6.1. CONSUMER HARDWARE

Both benchmarks, the CockroachDB and go-web-framework-benchmark, are executed on consumer hardware. This experiment will show whether there are improvements to be made on consumer systems running conventional hardware and a consumer operating system.

6.1.1. COCKROACHDB

Looking at the results for CockroachDB in Figure 6.1, there is an improvement by using the bidirectional hillclimbing method over the default `GOMAXPROCS`. Performance has increased by 5%, while also being more consistent. The opposite has happened with the linear search method. It has its performance decreased by 12%. This big regression is caused by one run where the operations performed was a third of the other runs. This is indicated by the increased standard deviation. The reason for this is that the value for `GOMAXPROCS` settled on a value far from the optimal, which happens with a small amount of runs. Increasing the amount of runs does not remedy this, selecting an optimal value is difficult for CockroachDB.

The reason for this bad run is caused by the spiky nature of the progress graph in Figure 5.1. The graph shows several dips in performance, where there is very little progress made. Due to internal CockroachDB processes or tasks, performance drops to almost

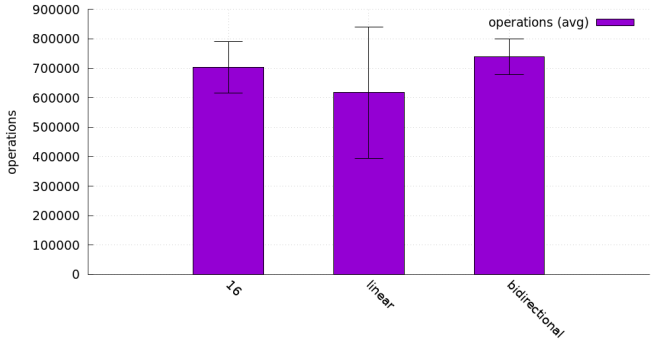


Figure 6.1: Operations on two instances of CockroachDB

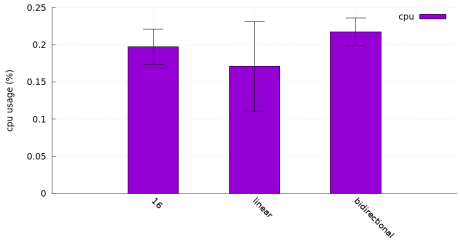


Figure 6.2: CPU usage, default vs alternatives, CockroachDB

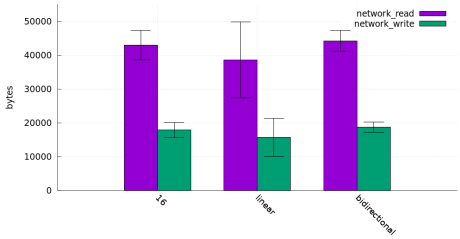


Figure 6.3: Network usage, default vs alternatives, CockroachDB

zero. When this happens during the search-phase, from 10 to 26 seconds, the best value for `GOMAXPROCS` is not chosen and a non-optimal value is picked. This results in poor performance, which is then locked until the linear search is started again. During testing, linear search is not started again, resulting in permanently lowered performance for the run. Excluding this one outlier, the linear search solution will outperform the fixed `GOMAXPROCS=16` configuration by 2%.

Fixing this issue can be done by extending the linear search time to a duration significantly longer to ensure that testing of a `GOMAXPROCS` value takes longer than the dip. In that case, it would settle on a value that is close to the optimal, and not far away. Alternatively, bounds can be introduced to limit the selection of `GOMAXPROCS` only to a range of values depending in the amount of Go schedulers started.

For the benchmarks executed with the custom kernel, CPU and network usage is tracked. Comparing the results from the CPU and network usage in figures 6.2 and 6.3 to the results from the operations performed in Figure 6.1, it shows that there is not any additional information taking into account these numbers. On consumer hardware, these CPU and network usage figures do not provide better information than the total amount of operations performed.

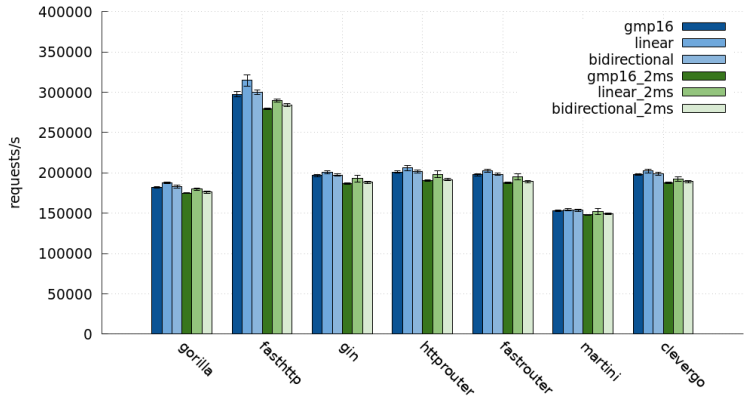


Figure 6.4: Performance for gwfb

6.1.2. GO-WEB-FRAMEWORK-BENCHMARK

Following the CockroachDB analysis, we turn to gwfb for the thread-per-request model. The results are displayed in Figure 6.4. The colors blue and green designate the split between the measurements with and without a 2ms sleep in each call. The default setting is outperformed in both types of measurements, with the linear search performing the best. There are differences between the various web framework implementations, but for each framework, the linear search and bidirectional hillclimbing methods outperform the default. Additionally, it can be observed that the 2ms runs perform a bit worse than the 0ms runs. This is an expected change, since the go scheduler schedules other calls while keeping track of sleeping calls. Once a sleep duration is done, the scheduler will still be working on a newer call, having to wait a bit to resume the just woken-up call. This results in a kind of buildup of in-progress calls in the system which the scheduler has to keep track of.

To make the results more insightful, we create an overview comparing the measurements to the default thread pool size. The gains of each algorithm compared to the default are shown as percentages in Figure 6.5. This analysis shows that the best performing solution is the linear search algorithm, where the `GOMAXPROCS` value is not changing after the search-phase. The improvement is an increase of 3% in requests per second over the default. The improvement for the bidirectional hillclimbing method comes in at 1%, although not as much as the linear search method, this algorithm is more suitable for changing workloads as shown in Figure 6.1.

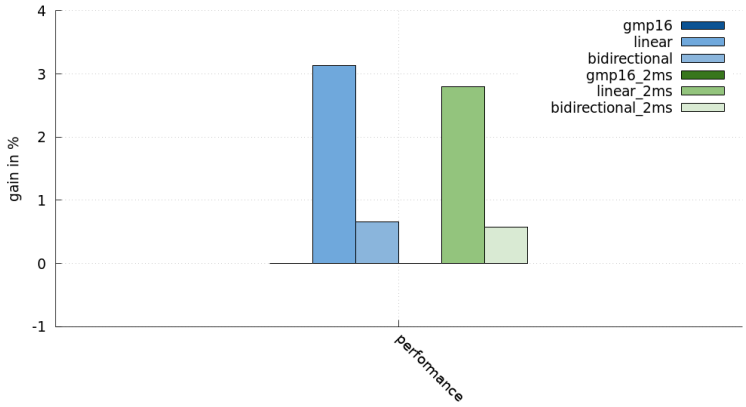


Figure 6.5: Percentage gains for requests per second for gwfb

6.2. SERVER HARDWARE

Our server benchmarks offer a bit more information about the system. As mentioned in section 4.1, we can track power consumption for nodes in the DAS6 cluster. With this information we can evaluate how the system performs in terms of energy usage over the different configurations. These benchmarks are split up in terms of execution, the applications to benchmark are run on different nodes as the load-generators. First up we will evaluate CockroachDB, after that the go-web-framework-benchmark.

6.2.1. COCKROACHDB

The results from CockroachDB on the DAS6 cluster contain a couple more figures compared to the locally run benchmarks. With benchmark execution on the DAS6 cluster, the power usage during the benchmark can be measured. These will be evaluated later, first we start with the basic performance numbers. In Figure 6.6, we can see that compared to the default, the linear search performs slightly worse, handing in 1.7% performance. Compared to that, the bidirectional hillclimbing method performs slightly better and gains 4.6% performance. This difference in performance is caused by the linear search encountering the same issues as it does locally, illustrated by Figure 5.1. With CockroachDB, there are moments where operations performed temporarily drops, which results in a non-optimal value for `GOMAXPROCS` being selected. This is an issue that is solved by the bidirectional hillclimbing method, which is able to adapt to the changing operations each second.

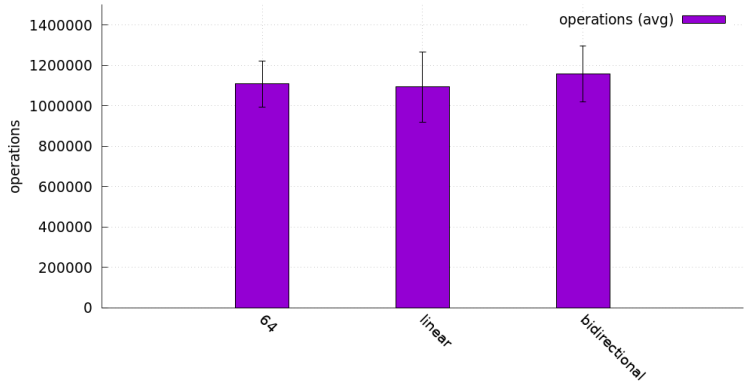


Figure 6.6: Performance for CockroachDB on DAS6

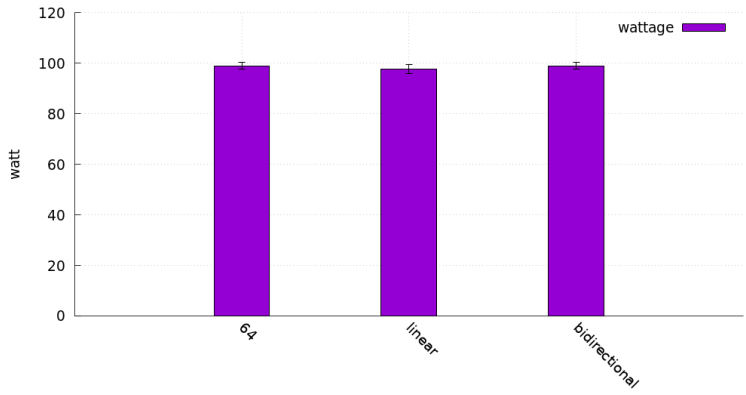


Figure 6.7: Power usage for CockroachDB on DAS6

Figure 6.7 shows power usage, which shows interesting results. It shows that the power usage is almost identical for all instances. The linear search has a decrease of 1.3%, while also performing -1.3% worse. The difference in operations and wattage remains the same. If we now look to the bidirecitonal hillclimbing result, it shows a power usage that increases by 0.1%, accompanied with a performance increase of 4.6%. While performing better, the wattage remained the same.

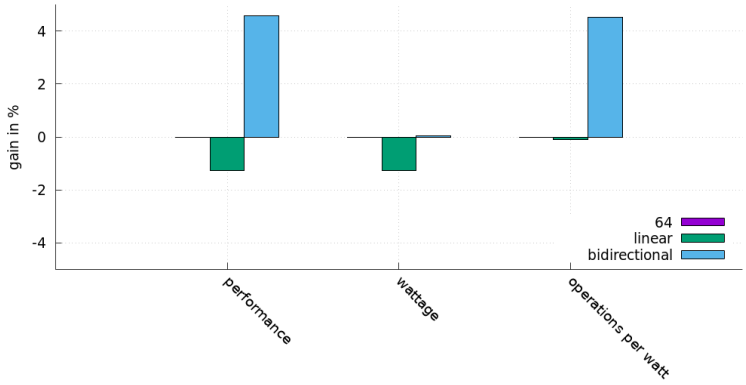


Figure 6.8: Percentage gains for CockroachDB on DAS6

	performance	wattage	operations per watt
64	0.00%	0.00%	0.00%
linear	-1.26%	-1.27%	-0.08%
bidirectional	4.58%	0.06%	4.52%

Table 6.1: characteristics versus default for CockroachDB on DAS6

The figure showing the operations per watt for these CockroachDB benchmarks are left out due to the heavy similarities to the results in Figure 6.6. Figure 6.8 and Table 6.1 show the improvements made in a more comprehensible view. It shows that in the end, the operations per watt remains the same for the linear search method, with a performance regression of -1.3%. Compare this to the operations per watt improvement of 4.5% and performance improvement of 4.6% of the bidirectional hillclimbing method. From these results, we can conclude that for the changing value of the CockroachDB operations per second, the continuously adapting bidirectional hillclimbing algorithm yields the best results.

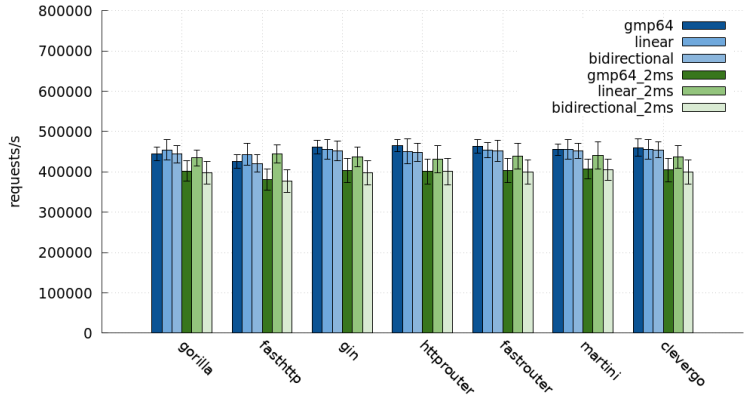


Figure 6.9: Performance for gwfb on DAS6

6.2.2. GO-WEB-FRAMEWORK-BENCHMARK

Now executing the gwfb on the DAS6 cluster allows us to run the benchmark in a distributed fashion. As previously mentioned, the DAS6 cluster allows measurements of power consumption for nodes running in the cluster, next to the performance metrics. Starting off with the performance of the frameworks in Figure 6.9, the results show that there are a couple differences between implementations. For hillcimbing, gorilla, fasthttp and martini, there is a small increase. However, for the rest of the frameworks in both linear search and bidirectional hillclimbing, perform slightly worse. On average, for the linear search, there is a performance regression of -0.1% and -1.6% for bidirectional hillclimbing. The big improvement takes place when Go has to handle more requests concurrently and the scheduler has to handle more threads. This can be observed in the `*_2ms` measurements for the linear search algorithm, with an overall improvement of 9.4% in performance. The bidirectional algorithm does not manage to improve, a decrease in performance of -0.8% can be observed.

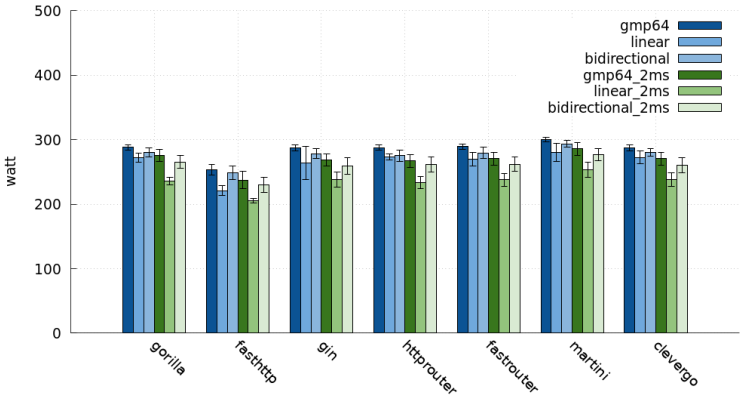


Figure 6.10: Power usage for gwfb on DAS6

	performance	wattage	requests/s per watt
gmp64	0.00%	0.00%	0.00%
linear	-0.10%	-7.17%	8.03%
bidirectional	-1.60%	-2.85%	1.27%
gmp64_2ms	0.00%	0.00%	0.00%
linear_2ms	9.40%	-12.48%	25.17%
bidirectional_2ms	-0.87%	-3.17%	2.37%

Table 6.2: Characteristics versus default for gwfb on DAS6

Results from enabling `likwid-powermeter` are displayed in Figure 6.10. The new methods for setting `GOMAXPROCS` show surprising improvements over the default. In each instance, the wattage of the benchmarks has decreased resulting in an average of -7.2% for linear search and -2.9% for bidirectional hillclimbing. With the `*_2ms` measurements, this extends to -12.5% and -3.2% for linear search and bidirectional hillclimbing respectively.

Combining these improvements into an aggregated requests per second per watt results in Figure 6.11. It becomes clear how much impact the linear search algorithm has on the performance and power efficiency. The overall decreased performance of -0.1% for the linear search algorithm, combined with the reduced power consumption of -7.2%, results in an increased requests per second per watt of 8.0%. For the `*_2ms` case, the requests per second per watt increased by 25.2%. These results are further clarified in Figure 6.12 and Table 6.2. The linear search algorithm loses a small amount of performance at worst and improves for the `_2ms` case, while vastly reducing the wattage of the executing machine.

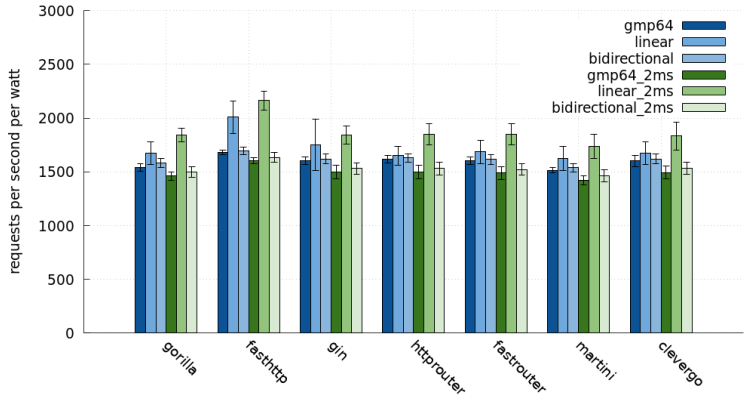


Figure 6.11: Performance per watt for gwfb on DAS6

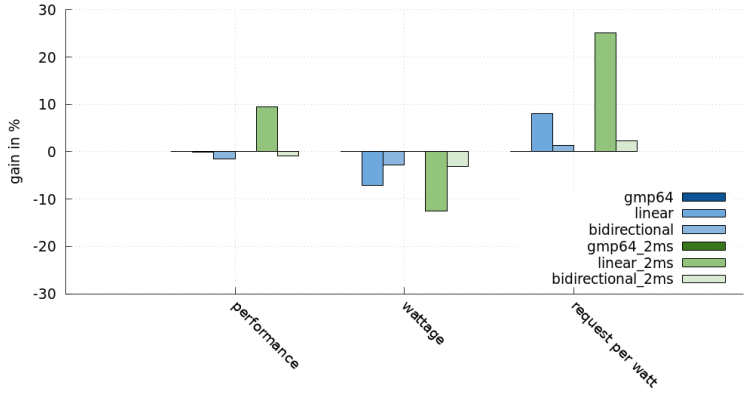


Figure 6.12: Percentage gains for requests per second per watt for gwfb on DAS6

7

CONCLUSION

7.1. CONCLUSION

In this thesis, several research questions are investigated to support and help in answering the main research question: *"Can we improve the performance of multiple programs on the same machine?"*. With the supporting material from previous sections, these will be answered below.

RQ1: Can adaptive thread pools be extended beyond disk I/O?
Concluding from the exploration and the measurements performed in chapter 4, improving performance using CPU metrics seems to be ineffective. It also became apparent that compared to network usage, operations completed was a more promising value to use for steering the thread pool. The operations completed has a larger percentual difference when performance varies, which makes steering the thread pool more predictable.

RQ2: What is the most effective way of collecting metrics for the control loop to act upon?
From the analysis performed in 2.3, combined with the successful implementation of in the benchmarks, it shows that taskstats is the best way to collect metrics, with the sole downside that network information had to be added to the taskstats interface 2.3.1. With this extension, thread-based data regarding CPU and network traffic are now available on a real-time basis, instead of being bundled for the entire program. This allows for more fine grained insights into the scheduler activity.

RQ3: Can the hillclimbing approach be improved to handle changes in variable workloads?
The hillclimbing approach can be improved for variable workloads. The bidirectional hillclimbing is able to improve the performance of CockroachDB, where a more dynamic approach is required. From observing the current implementations, the bidirectional

hillclimbing implementation works best in a constantly changing environment. Opposite to what the proposed hillclimbing implementation does [22], this implementation immediately reacts to changes in performance and attempts to find the new optimal value for the thread pool.

RQ: Can we improve the performance of multiple programs on the same machine? Both on consumer and server hardware, concurrently deployed applications can have their performance improved by scaling the thread pool by steering on operations.

For consumer hardware, performance can be improved, although it does depend on the application which algorithm is best suited. CockroachDB has been improved by 5% using the bidirectional hillclimbing method, while the linear search performed worse than the default. The go-web-framework-benchmark can be improved by 3% with linear search and by 1% with bidirectional hillclimbing.

For server hardware, big improvements are realized. Additionally, with the ability to measure power usage, there is an overall power decrease when scaling the thread pool. This method is quite effective, improving performance per watt while in the worst cases incurring only a small performance penalty. Running CockroachDB on the DAS6 cluster resulted in a 5% performance increase and 5% more operations per watt using the bidirectional hillclimbing method. For the go-web-framework-benchmark, linear search achieved a 9% performance increase and a 25% improved requests per watt.

Additionally, we have created a new monitoring system that fills the gap between the ad-hoc monitoring tools and the debugging-oriented tools. This monitoring system is able to track all data that is currently exposed through taskstats, including network system calls and transferred data, on a per-thread basis. This type of monitoring solution allows for further examination and improvement of existing thread pool solutions through the ability of isolated thread-graphing.

7.2. RECOMMENDATIONS

From the results this research has produced, there are subsequent steps to be made to further progress this subject. Directly associated to this research, the algorithms that select the thread pool size can be improved.

From the observations in section 5.1.4, the linear search and bidirectional hillclimbing methods can be improved to handle the drops in performance which occasionally occur (as seen in Figure 5.1). These drops should not influence the result of the search-phase. A mechanism in the algorithms to observe these drops and not let them affect the search phase or the hillclimbing would improve the results greatly and further reduce standard deviation. For the bidirectional hillclimbing algorithm, these drops are less noticeable, although the algorithm does still need a short time to recover from these sudden decreases in performance.

Taken from that same section 5.1.4, the bidirectional algorithm can be improved by

a mechanism that eliminates the constant switching between two values, however, the continuous searching behavior cannot and should not be eliminated. The continuous searching behavior makes the algorithms suitable for dynamic environments.

Apart from the improvements that can be made directly to the contributions in this research, there are other actions to advance on this research. Starting off, the improvements presented here need to be validated using other thread pool implementations. Since we only take into account the Go programming language, there are other programming languages where thread pool scaling might not work. Rust, Java and Python all have libraries implementing thread pools which are utilized in other products. Once the solutions are validated in other programming languages, a mixed implementation can be tested, validating by combining programming languages or libraries with each other. This mixed implementation could confirm whether or not a more universal approach would be equally effective in improving performance and efficiency.

Alternatively, there is an option to create a thread pool implementation in the Linux kernel and let that thread pool handle all decisions, instead of each separate thread pool working independently. A solution like this has been shown successful for Grand Central Dispatch in iOS and OS X. Combined with the results from this research, a centralized thread pool this can potentially improve performance and provide energy savings over multiple programs.

BIBLIOGRAPHY

[1] Yibei Ling, Tracy Mullen, and Xiaola Lin. “Analysis of optimal thread pool size”. In: *ACM SIGOPS Operating Systems Review* 34.2 (2000), pp. 42–55.

[2] Shiping Chen and Ian Gorton. “A predictive performance model to evaluate the contention cost in application servers”. In: *Ninth Asia-Pacific Software Engineering Conference, 2002*. IEEE. 2002, pp. 435–440.

[3] Nathan Tuck and Dean M Tullsen. “Initial observations of the simultaneous multithreading Pentium 4 processor”. In: *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2003, pp. 26–34.

[4] Dongping Xu. *Performance study and dynamic optimization design for thread pool systems*. Tech. rep. Ames Lab., Ames, IA (United States), 2004.

[5] Ji Hoon Kim et al. “Prediction-based dynamic thread pool management of agent platform for ubiquitous computing”. In: *International Conference on Ubiquitous Intelligence and Computing*. Springer. 2007, pp. 1098–1107.

[6] DongHyun Kang et al. “Prediction-based dynamic thread pool scheme for efficient resource usage”. In: *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*. IEEE. 2008, pp. 159–164.

[7] Allan Porterfield, Rob Fowler, and Mark Neyer. “Maestro: Dynamic runtime power and concurrency adaptation”. In: *Proc. Workshop Managed Many-Core Syst.* Citeseer. 2008, pp. 1–8.

[8] Chee Siang Wong et al. “Towards achieving fairness in the Linux scheduler”. In: *ACM SIGOPS Operating Systems Review* 42.5 (2008), pp. 34–43.

[9] Taylor Groves, Jeff Knockel, and Eric Schulte. “Bfs vs. cfs scheduler comparison”. In: *The University of New Mexico* 11 (2009).

[10] Brian F Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[11] Robert Hood et al. “Performance impact of resource contention in multicore systems”. In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.

[12] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. “Addressing shared resource contention in multicore processors via scheduling”. In: *ACM Sigplan Notices* 45.3 (2010), pp. 129–142.

[13] Kang-Lyul Lee et al. “A Novel Predictive and Self–Adaptive Dynamic Thread Pool Management”. In: *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2011, pp. 93–98.

[14] Kazuki Sakamoto and Tomohiko Furumoto. “Grand central dispatch”. In: *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 2012, pp. 139–145.

[15] Allan K Porterfield et al. “Power measurement and concurrency throttling for energy reduction in openmp programs”. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 884–891.

[16] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 2014, pp. 305–319.

[17] H. Bal et al. “A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term”. In: *Computer* 49.05 (May 2016), pp. 54–63. ISSN: 1558-0814. DOI: 10.1109/MC.2016.127.

[18] Jean-Pierre Lozi et al. “The Linux scheduler: a decade of wasted cores”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. 2016, pp. 1–16.

[19] Thomas N Theis and H-S Philip Wong. “The end of moore’s law: A new beginning for information technology”. In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50.

[20] Alexandru Uta and Harry Obaseki. “A performance study of big data workloads in cloud datacenters with network variability”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 113–118.

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

[22] Jannes Timm. “An OS-level adaptive thread pool scheme for I/O-heavy workloads”. In: (2021).

[23] Jannes Timm and Jan S Rellermeyer. “Why Multi-Threading Should No Longer Be a DIY Job”. In: *Tagungsband des FG-BS Frühjahrstreffens 2022* (2022).

[24] CockroachDB. *Architecture Overview | CockroachDB Docs*. URL: https://www.cockroachlabs.com/docs/stable/architecture/overview.html%5C#goals-of-cockroachdb. (accessed: 09-03-2022).

[25] CockroachDB. *cockroach workload | CockroachDB Docs*. URL: https://www.cockroachlabs.com/docs/v21.2/cockroach-workload.html%5C#workloads. (accessed: 09-03-2022).

[26] Linux kernel. *CFS Scheduler – The Linux kernel documentation*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html. (accessed: 11-02-2022).

[27] *linux/taskstats.h at master*. URL: https://github.com/torvalds/linux/blob/master/include/uapi/linux/taskstats.h. (accessed: 28-02-2022).

[28] Intel Newsroom. *Introducing 12th Gen Intel Core Processors*. URL: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html. (accessed: 13-01-2022).

[29] *Per-task statistics interface*. URL: https://www.kernel.org/doc/Documentation/accounting/taskstats.txt. (accessed: 28-02-2022).

[30] Allied Market Research. *Global Microservices Architecture Market to Reach $8.07 Billion by 2026: Says, Allied Market Research*. URL: https://www.globenewswire.com/news-release/2020/11/09/2122841/0/en/Global-Microservices-Architecture-Market-to-Reach-8-07-Billion-by-2026-Says-Allied-Market-Research.html. (accessed: 23-05-2022).

[31] *Scaling in the Linux Networking Stack*. URL: https://www.kernel.org/doc/html/latest/networking/scaling.html. (accessed: 05-04-2022).

[32] *Sched(7) - Linux manual page*. URL: https://man7.org/linux/man-pages/man7/sched.7.html. (accessed: 20-07-2022).

[33] *Scheduling Priorities - Win32 Apps*. URL: https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities. (accessed: 28-07-2022).

[34] *socket(2) - Linux manual page*. URL: https://man7.org/linux/man-pages/man2/socket.2.html. (accessed: 17-03-2022).

[35] Passmark Software. *PassMark Software - Hardware Survey - CPU Cores*. URL: https://www.pcbenchmarks.net/number-of-cpu-cores.html. (accessed: 24-05-2022).

[36] *Sysfs(5) - Linux manual page*. URL: https://man7.org/linux/man-pages/man5/sysfs.5.html. (accessed: 31-05-2022).

A

FULL STRACE RESULTS

Listing A.1: strace results from CockroachDB instance

% time	seconds	usecs/call	calls	errors	syscall
62,16	32,559433	18	1791011	40066	futex
12,33	6,460259	85	75431		fdatasync
9,15	4,791918	2	1808833	94	rt_sigreturn
8,29	4,344176	16	267021	3	nanosleep
5,74	3,009010	21	139454		epoll_pwait
1,15	0,603630	3	167650		write
0,37	0,196342	12	16040		pread64
0,31	0,161232	2	58415	28521	read
0,16	0,083747	15	5255		getpid
0,13	0,070356	12	5843		sched_yield
0,05	0,025620	4	5203		tgkill
0,04	0,018851	2	7515	2083	newfstatat
0,03	0,015666	7	2026		madvise
0,03	0,014066	7	1835	785	openat
0,02	0,009409	6	1445	1438	unlinkat
0,01	0,004035	1	2074	1721	epoll_ctl
0,00	0,002339	2	1060		getdents64
0,00	0,001904	32	58		sync_file_range
0,00	0,001879	1	1051		close
0,00	0,000870	72	12		fsync
0,00	0,000617	1	357		fcntl
0,00	0,000257	3	83		mmap
0,00	0,000255	1	217		fstat
0,00	0,000074	74	1		waitid
0,00	0,000061	3	20		fstatfs

A

0,00	0,000041	2	15		poll
0,00	0,000040	0	69		statfs
0,00	0,000028	4	6		clone
0,00	0,000021	1	13	4	connect
0,00	0,000019	1	12		socket
0,00	0,000017	5	3		pipe2
0,00	0,000011	1	6		munmap
0,00	0,000010	2	5		sendmmsg
0,00	0,000010	0	13		getrandom
0,00	0,000009	0	26		rt_sigprocmask
0,00	0,000007	0	15		mprotect
0,00	0,000007	0	11		ioctl
0,00	0,000007	0	10		recvfrom
0,00	0,000007	7	1		wait4
0,00	0,000007	0	29		fadvise64
0,00	0,000005	5	1		epoll_create1
0,00	0,000004	0	6		lseek
0,00	0,000003	1	3		uname
0,00	0,000003	3	1		getuid
0,00	0,000002	0	11		setsockopt
0,00	0,000001	0	5		renameat
0,00	0,000001	0	2		fallocate
0,00	0,000000	0	1		open
0,00	0,000000	0	2		brk
0,00	0,000000	0	120		rt_sigaction
0,00	0,000000	0	1	1	access
0,00	0,000000	0	1		sendto
0,00	0,000000	0	3		recvmsg
0,00	0,000000	0	1		bind
0,00	0,000000	0	4		getsockname
0,00	0,000000	0	1		getpeername
0,00	0,000000	0	1		execve
0,00	0,000000	0	1	1	readlink
0,00	0,000000	0	2		umask
0,00	0,000000	0	2		sigaltstack
0,00	0,000000	0	2	1	arch_prctl
0,00	0,000000	0	1		gettid
0,00	0,000000	0	2		sched_getaffinity
0,00	0,000000	0	1		set_tid_address
0,00	0,000000	0	1		readlinkat
0,00	0,000000	0	1		set_robust_list
0,00	0,000000	0	1		prlimit64
100,00	52,376266	12	4358322	74718	total

B

GO WEB FRAMEWORK BENCHMARK

Listing B.1: benchmarking options in go-web-framework-benchmark

```
web_frameworks=("default" "atreugo" "beego" "bone" "chi" "clevergo"
                "denco" "echo" "fasthttp" "fasthttp-routing"
                "fasthttp/router" "fasthttprouter" "fastrouter"
                "fiber" "flygo" "fresh" "gear" "gearbox" "gin"
                "goframe" "goji" "gojsonrest" "gongular" "gorestful"
                "gorilla" "gorouter" "gorouterfasthttp" "go-ozzo"
                "gowww" "goyave" "httprouter" "httptreemux" "lars"
                "lion" "martini" "muxie" "negroni" "neo" "pat" "pure"
                "r2router" "tango" "tiger" "tinyrouter" "traffic"
                "treemux" "violetear" "vulcan" "webgo")
```