Bachelor's Thesis
(Course Code: XB_40001)

Using the unikernel development kit "Unikraft" to develop networked embedded systems applications

by

Job Paardekooper
(Student Number: 2736300)

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

July 19, 2024

July, 2024, Vrije Universiteit Amsterdam

Certified by .....
Cristiano Giuffrida
Associate Professor
First Supervisor

Certified by .....
Asia Slowinska
Assistant Professor
Second Reader

Using the unikernel development kit "Unikraft" to develop networked embedded systems applications

Job Paardekooper
Vrije Universiteit Amsterdam
Amsterdam, NL
j.w.paardekooper@student.vu.nl

ABSTRACT

Unikraft is a unikernel development kit aiming to simplify unikernel development by focusing on componentization, configurability and tooling integration. While primarily used in cloud environments, Unikraft also offers potential benefits for embedded systems. Unikraft's POSIX compatibility allows it to run Linux applications as efficient unikernels. Running these unikernels on embedded systems opens the door to optimizations such as reduced memory usage, fast boot times, improved application performance and more.

In this thesis, we benchmark TCP and Nginx throughput running on an improved version of the Unikraft Raspberry Pi (RPi) platform. We updated the Unikraft RPi platform to the latest version, integrated a network driver, and compared its performance with a similar Linux setup. Our benchmarking of Nginx on Unikraft with a single connection shows up to a 25% improvement in requests per second compared to Linux. However, due to limitations in the current implementation, Unikraft's maximum TCP throughput is 16% lower than Linux, and with 10 connections, Nginx on Unikraft is 20%-78% slower compared to Linux.

Source code and raw measurement data used in this thesis can be found at www.github.com/jobpaardekooper/unikraft-rpi.

1 INTRODUCTION

Unikernels, specialized single-purpose operating systems, offer significant improvements in performance and resource efficiency through their tailored design [19]. They combine the operating system and user space application into a single program, allowing developers to fine-tune their applications at the right abstraction level. This approach reduces the size of the trusted computing base and frees up runtime resources by omitting unnecessary OS functionalities. Despite these advantages, porting applications to unikernel environments has historically been time-consuming due to compatibility and development challenges [19].

The Unikraft project [7] is a unikernel development kit that aims to address the main obstacles in unikernel development. By providing a POSIX-compliant environment, Unikraft significantly reduces the time required to port applications [19]. It introduces a kernel built around micro-libraries, each providing specific OS primitives. The micro-library approach allows developers to select only the necessary components for their applications. This modular approach facilitates performance tuning and reduces porting times.

The Internet of Things (IoT) represents a growing market [6], where the optimizations provided by unikernels are particularly promising. Embedded IoT devices commonly have significant constraints on computational and storage resources. Embedded systems frequently run a specialized single-application with minimal to no operating system abstraction. These characteristics align closely with unikernels, as they are minimal, resource-efficient, single-application operating systems. Therefore, unikernels and Unikraft suggest a natural avenue to explore in the context of embedded systems. Aditionally, Unikraft's ability to efficiently run off-the-shelf Linux applications [28] in a minimal environment enhances the security of IoT systems by reducing the trusted computing base which reduces the attack surface [19]. Furthermore, by making it easier to run more complex applications directly on IoT devices rather than relying on cloud-based processing, Unikraft reduces latency, improves responsiveness, and eases bandwidth and privacy concerns associated with data transmission to the cloud. Finally, embedded devices often interface with hardware peripherals and so by removing the costly barrier between user and kernel space, the overhead of interacting with hardware is reduced.

In this thesis, we investigate how Unikraft can benefit the bare-metal deployment of networked applications on embedded systems and present the following key contributions:

(1) **Improvement of the Unikraft RPi platform:** We have updated the Unikraft target platform for the Raspberry Pi 3 Model B (RPi 3B) to provide compatibility with the latest Unikraft version (*v0.16.3*). The RPi 3B was chosen due to the vast amount of resources available, as it is widely used by hobbyists. Additionally, the RPi 3B closely resembles the RPi 3B+, which was used in prior research upon which our study builds [28].
(2) **Network driver integration:** We have integrated a network driver based on the open-source USPi project [30] combined with Lightweight IP (lwIP) [14] to support the TCP/IP network stack.
(3) **Performance benchmarking:** We benchmarked the TCP and Nginx throughput of Unikraft against a comparable Linux setup using Raspberry Pi OS Lite (on a single core and with RamFS, for fair comparison).

Our results demonstrate Unikraft's potential to improve the performance of embedded networked applications while revealing additional areas where further research and improvement are needed. Specifically, our experimental evaluation provided the following insights:

- Benchmarking Nginx on Unikraft with a single connection shows improvements of up to 25% of average requests per second compared to Linux.
- Benchmarking Nginx with 10 simultaneous connections results in Unikraft being 20%-78% slower when compared to Linux.
- TCP benchmarking shows that Unikraft's maximum TCP throughput is 16% lower than Linux.

The current most important limitations specific to the Unikraft RPi platform include the fact that there is no multi-core support. Even though Unikraft does support multi-core processing, this has yet to be implemented on the RPi platform. Further, many embedded processors are based on 32-bit architectures, which is not supported by Unikraft.

2 BACKGROUND

In this section, we provide an overview of the foundational works and relevant research that support this study, highlighting key contributions that we have directly built upon and giving insight into the broader context surrounding this work.

2.1 Unikernels

Unikernels are specialized operating systems tailored to run a single application by combining the kernel and user space into a single, application-specific operating system. Several unikernel projects have been introduced over the years, including MirageOS, IncludeOS, OSv, LynxElement, and Unikraft [12, 18, 19, 25, 33]. Common ideas appearing in various unikernel projects include:

- **Application specific**: With unikernels being built to run only a specific application, their runtime can be optimized by dropping unused functionalities. Only essential functionalities required by applications from the kernel are kept. This introduces resource usage and performance optimizations. Additionally offering a reduced attack surface by including just the strictly necessary code, significantly reducing the size of the trusted computing base. This is a key reason why security-critical organizations such as Navies, Air Force and Armies are running unikernel trials [32].
- **Single protection level**: Using a single protection level eliminates the need for context switching between user and kernel modes inside unikernels, thus improving performance and providing direct hardware access. This enables hardware-specific optimizations and reduces the overhead associated with hardware I/O in traditional monolithic kernels.
- **Deployment targets for different hypervisors**: Traditional cloud virtual machine (VM) environments contain duplication of responsibilities between the hypervisor and the guest operating system kernel. Both are designed for tasks such as isolation and hardware resource management. Unikernels reduce this duplication by viewing hypervisors as platforms for running isolated applications (unikernels) instead of isolating full machines (VMs).

Most projects achieve considerable performance gains in multiple of the following areas:

- Fast boot times
- Small images sizes
- Reduced memory consumption
- Reduced system call overheads

2.2 Unikraft

In 2019, researchers introduced the Unikraft open-source project [7, 9, 20] to automate unikernel building, reducing development and porting times, while improving application performance. Later, Unikraft branded itself as a "Unikernel Development Kit". A detailed paper on Unikraft was presented in 2021 [19], revealing further details about its architecture and quantifying many of the performance gains made possible by it. Since these papers, the project has evolved significantly, adding support for different hypervisors and expanding POSIX API compatibility. These papers largely focused on cloud computing, leading to the recent creation of a cloud platform built fully around Unikraft and its unikernels [17].

To achieve its goals, Unikraft's unique design principles are focused on a fully modular kernel with minimal dependencies among components. Libraries provide OS primitives and conform to a well-defined API, allowing multiple libraries to solve problems in different ways [19]. Developers can select different types of allocators, schedulers and other operating system components, making it easy to test different implementations and determine the exact right solution for their needs. Platform support in Unikraft is also designed around libraries, where code can be reused across platforms by abstracting it into common modules. This reduces the time and complexity involved in adding support for new platforms.

To aid application porting, Unikraft supports a binary compatibility mode [19], allowing developers to run Linux applications without recompilation. This provides a quick way for developers to estimate (a lower bound) for performance gains they can achieve when porting their application to Unikraft. Quickly testing applications using binary compatibility also generates insights into how well the current applications are supported by Unikraft (since Unikraft is still under development). Currently, this binary compatibility mode is limited to x86 architectures.

These features of Unikraft are significant reasons to investigate it in this study. Additional reasons include:

- Unikraft does not focus on a specific language, unlike MirageOS and IncludeOS, which focus on OCaml and C++ respectively [12, 25].
- Unikraft is open-source, unlike the proprietary LynxElement [33].
- Unikraft has existing research into its application on embedded systems [28], unlike OSv, which focuses primarily on cloud deployments [18].
- Unikraft is under very active development at the time of this study [7].

2.3 Unikernels on embedded systems

Many of the unikernel projects mentioned previously focus on the cloud as their primary use case, viewing unikernels as a potential replacement for containers in the cloud. But projects like IncudeOS [27], LynxElement [33] and Unikraft [28] also explore the potential for unikernels on embedded systems.

In 2018, IncludeOS announced plans to support embedded ARM64-based SoCs, starting with the Raspberry Pi 3 Model B+ [27]. However, this platform has not been released for public use. Additionally, IncludeOS development has slowed since 2019. It saw no releases and a total of three commits between March 2020 and March 2024 [1]. The current state and future of IncludeOS and its bare-metal support are unclear.

LynxElement's proprietary nature limits public information about its use on embedded systems, and without access to the source code, it is not an option that can easily be explored further.

This leaves Unikraft as a promising option for embedded systems. In 2020 a work-in-progress paper was introduced that explored Unikraft's potential in embedded systems [28]. It introduced and benchmarked an initial Unikraft platform for the Raspberry Pi 3 B+, demonstrating Unikraft's potential for ARM64-based embedded devices. The study focused on measuring boot time, power usage and memory consumption while running applications like Nginx. Results showed boot times between 88 and 158 milliseconds, with memory consumption in the hundreds of kilobytes. Support for POSIX facilitates running off-the-shelf Linux applications on constrained embedded devices, avoiding the potentially unacceptable performance of a full Linux installation. Additionally, it allows for a more developer-friendly environment compared to traditional bare-metal coding. This allows for the use of familiar tools and libraries, thus significantly easing the development process.

The newly introduced platform did not include a network driver, so no performance testing was accomplished while the applications were running. In this study, we update and improve the code introduced for the Raspberry Pi platform and quantify the runtime performance of Nginx on Unikraft compared to Linux.

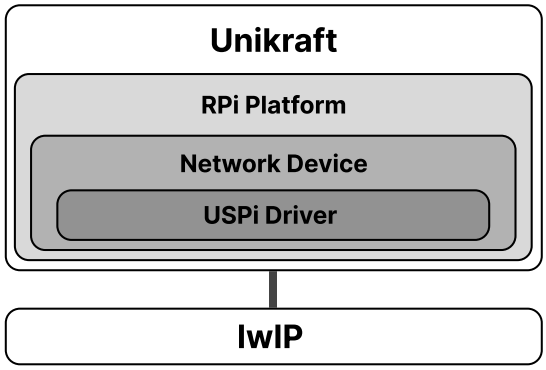


Figure 1: ***Visual overview of how the various components of the system fit together. The USPi and network device are part of the internal Unikraft RPi platform. lwIP functions as an external library, interfacing with Unikraft's "public" API to complete the full networking stack.***

3 OVERVIEW

This section presents an overview of the system, its main components, and how they integrate. The main goal of the system is to run networked applications inside a Unikraft unikernel on the Raspberry Pi 3 Model B. To achieve this, four main components are combined. The Unikraft RPi platform, a networking device for the RPi inside Unikraft, the USPi ethernet driver, and the LightWeight IP networking stack.

Figure 1 shows a visual overview of how these components fit together. Unikraft contains the RPi platform library which includes the network device based on USPi. We use lwIP as an external library that completes the network stack by providing a TCP/IP implementation to Unikraft, providing our test applications with all the necessary components to run.

3.1 Unikraft Raspberry Pi 3 Model B target platform

We start from the existing Unikraft RPi Platform originally developed for the RPi 3B+ [28]. Platforms in Unikraft are structured as internal libraries. The Unikraft RPi platform has to implement specific functionalities expected from a platform library. These main functionalities include:

- Production of a unikernel image for the RPi using a linker script and Makefile.
- Configuration of the interrupt vector table, Memory Management Unit (MMU), and its page tables using ARM64 initialization code.
- Registration of the memory map into Unikraft to inform it about usable memory regions.
- Implementation of a serial console.
- Implementation of a timer.

In the remainder of this thesis we will use "RPi platform" to refer to the Unikraft RPi platform library. We support and work with the RPi 3B. The original platform was tested on the slightly different RPi 3B+. These devices mainly differ in CPU clock speed and some peripherals [24]. Since the two SoCs are nearly identical, no code in the base Unikraft RPi platform needs to be adapted to accommodate this difference.

3.2 USPi ethernet driver

USPi is an open-source project for the RPi implementing a USB driver [30]. Since the ethernet port of the RPi 3B is accessed internally via USB, we use USPi to interact with the ethernet hardware [31].

3.3 Raspberry Pi network device

To integrate the USPi driver with Unikraft, we wrap USPi into a network device (netdev) conforming to the standard netdev API used inside Unikraft. This netdev is part of the Unikraft RPi platform, bridging Unikraft and the USPi driver.

3.4 LightWeight IP

LightWeight IP (lwIP) is an open-source network protocol stack (including TCP/IP) specifically designed with embedded systems in mind [14]. The Unikraft organization provides a slightly modified version of lwIP that integrates with Unikraft as an external library [10]. We utilize lwIP to provide DHCP, UDP, and TCP implementations, enabling applications to use POSIX sockets and other networking features as expected.

4 DESIGN & IMPLEMENTATION

This section covers the design and implementation of the RPi platform that supports networking for the RPi 3B. We also cover some of the major limitations while leaving a more detailed discussion on limitations for section 6.

4.1 Updating the Unikraft Raspberry Pi platform

Unikraft has undergone many changes since the RPi platform was released, we updated the platform to work correctly with the latest version of Unikraft (*v0.16.3*).

Unikraft needs descriptors of the various memory regions made available by the platform. Previously, the platform library needed to provide a function to Unikraft that returned a description of the memory regions. The platform defined this function and Unikraft would call it when needed, now the platform needs to actively register its memory regions. Additionally, the format of the memory region descriptors has slightly changed. We updated the platform to proactively register the memory region descriptors conforming to the new format during RPi platform initialization.

Unikraft used to also call IRQ system setup using a function provided by the platform. We updated the platform to automatically call the IRQ system setup during platform initialization. In section 6, we discuss why the IRQ system of the RPi platform should be restructured in the future.

We applied changes to the Makefile and linker script to correctly include common platform code that has moved to a different place in the Unikraft codebase. Additionally, the Makefile now includes code from the common Unikraft ARM64 platform to correctly enable floating point operations.

The original RPi platform had the ARM caches disabled for simplicity reasons. In the updated platform, we enabled the CPU caches to significantly improve performance. Enabling instruction caches was straightforward since the current setup does not do any dynamic code loading. We set the instruction cache bit in the EL1 System Control Register (SCTLR_EL1) and configured the appropriate caching flags for the page tables.

Enabling data caches required more effort. We set the data cache bit in SCTLR_EL1 and added cache management instructions to the assembly start code of the RPi platform. This cache management is based on a combination of the existing Unikraft ARM64 KVM platform initialization code and code from the USPi project. Finally, we added cache management to the mailbox [11] code. Mailboxes enable the ARM and the VideoCore (VC) of the RPi to communicate. This is used to set up the UART serial console. Since the VC is an external device from the point of view of the ARM we need to make sure the data we want to communicate to it is properly written to the main memory. Unikraft already provides ARM64 cache cleaning helper functions so we were able to use these to clean the correct data address ranges holding the mailbox request data. With these changes, we enabled both instruction and data caches on the ARM, making the platform fully functional.

Multi-core support for the RPi platform has not been implemented. We consider this limitation during benchmarking in section 5.

4.2 Ethernet driver implementation

The USPi project is an open-source project created from the USB driver code originating from the Circle project. Circle is an open-source bare metal programming environment for RPi devices [29]. The USB code relies on the following systems that are included with USPi:

- Exception handling
- Display management
- Memory management
- Logging system
- Interrupt handling (IRQ)
- Timer management

When incorporating USPi into Unikraft, we removed or replaced several of its systems to avoid duplication and improve compatibility. The RPi platform already implements exception handling so this duplicate functionality was removed from the USPi code. Similarly, the display system was removed as it provided no functionality for the ethernet driver code. USPi's memory management system, which provided malloc and free functions, was removed in favor of Unikraft's existing memory allocation functions. The logging system, which originally wrote messages to UART and the display, was replaced with printf calls, leveraging the RPi platform's existing UART printing capability. These changes streamlined the integration and reduced redundancy in the codebase.

USPi included its own IRQ system for handling timers and interrupts from the USB controller. We merged this IRQ system into the existing RPi IRQ system.

The timer system uses a different timer from the default one which is used in the RPi platform. To avoid having to change many parts of the code base, this timer was left intact. The only change we made was to make sure it was properly connected to the updated IRQ system. In the future, this timer should be replaced.

USPi also uses the mailbox to communicate with the VC. For simplicity, this mailbox code was left in even though this introduces duplication. Since the RPi platform memory regions differ from the ones used in USPi, we added some caching code to the mailbox communication. USPi expects a non-cachable memory region to be available at a certain fixed address. This region does not exist in the RPi platform so we replaced it with the appropriate cache management instructions. In the future, this mailbox communication should be unified with the existing solution present in the platform.

With these changes, the RPi platform now compiles with this driver included. It cannot be used by applications since it does not yet present itself to Unikraft.

4.3 Network device implementation

To properly integrate USPi into Unikraft, we implemented a Unikraft network device (netdev). This RPi netdev wraps USPi into an API that Unikraft is familiar with. The new netdev was created based on the existing Virtio netdev for the KVM platform.

Due to the API offered by USPi, the netdev is only available in polling mode. Implementing support for an interrupt mode would take a significant amount of work because the interface provided by USPi was not created with Unikraft in mind. Some additional optimizations like TCP checksum offloading are also not implemented. Despite these limitations, performance is still very reasonable as will be discussed in the next section.

With this RPi netdev implemented, we can use lwIP on the Unikraft RPi platform. Applications can utilize lwIP networking just like they would on any other Unikraft platform.

5 EVALUATION

We evaluate the RPi platform by comparing Unikraft and Linux configurations through benchmarking a simple TCP program and the Nginx web server [15]. The performance metrics discussed in this section, like requests per second, are usually not the top priority when considering embedded systems development. However, performing these kinds of benchmarks helps to reveal valuable insights into the state of the platform and aids in identifying areas of improvement.

5.1 Hardware configuration

We ran the experiments described in the following subsections on a Raspberry Pi 3 Model B V1.2 2015. The ethernet port on this device is specified at 100 Mbps. The device on which the other side of the testing code ran was a 2021 16-inch 32 GB M1 Pro MacBook running macOS *14.3.1*. We connected these devices using a Cat5e ethernet cable combined with a 1 Gbps USB-C to ethernet adapter on the MacBook side. We enabled the connection through the macOS internet sharing feature with the MacBook connected to a wifi network. Linux and Unikraft both use this exact hardware setup while running the benchmarks.

5.2 Software configuration

To benchmark TCP throughput, we created a simple client/server application in C. The server runs on the MacBook and waits for a connection from the RPi. The RPi transfers a buffer containing randomly generated data to the MacBook. The MacBook records the time between receiving the first data from the socket until it reads the expected amount of bytes. This test is repeated a fixed number of times using the same connection.

To benchmark Nginx, we use WRK [16] version *4.2.0* on the MacBook to repeatedly request different files from the web server. Nginx serves a folder with lorem ipsum *.txt* files ranging from 64 B to 256 kiB in size. We use version *1.15.6* of Nginx. This is already an older version at the time of testing. We use this version because Unikraft's *lib-nginx* is based on this version.

5.3 Unikraft configuration

The Unikraft configuration to run the TCP client uses Unikraft's *hello-world* app as a base. From this base, we create a *workdir* to include all needed dependencies. Here we put Unikraft *v0.16.3* with the added RPi platform. We additionally add *lib-lwip* and *lib-musl*. The *lib-lwip* library is Unikraft's fork of lwIP as previously discussed. Musl [4] is an implementation of the C standard library with *lib-musl* being Unikraft's fork. The main file of the *hello-world* program is replaced with the TCP client test application.

We configure Unikraft using make menuconfig, selecting the RPi platform as the target and disabling debug options. For *lib-lwip* we enable DHCP and disable the loopback interface.

To test Nginx, we use Unikraft's Nginx example app as a base. The general configuration for this application is the same as above with some additions. Nginx needs a file system that contains the configuration files along with the *.txt* files it needs to serve. To provide this, we configure a compiled-in and in-memory filesystem. The file system is provided in the configuration in *.cpio* format and gets compiled into the output image. The Nginx code is provided in the form of Unikraft's *lib-nginx* library. For the unikernel to start Nginx as the main program we need to select the "provide main function" option inside the *lib-nginx* menuconfig options.

Both exact *.config* result files used for testing these applications can be found in the GitHub repository [26], with more detail being provided in the appendix 7. Building these applications results in a *kernel8.img* output file.

To run this image on the RPi, we needed a properly formatted SD card containing the firmware and supporting files. We created this SD card by using the official Raspberry Pi Imager tool. We selected Raspberry Pi OS Lite to be written to the SD card. To run our unikernel applications we can simply replace the existing *kernel8.img* in the boot partition of the SD card with the one generated by the build process. Finally, we added some additional configuration options to the *config.txt* file to enable UART, 64-bit mode, and disable some unnecessary boot delays.

5.4 Linux configuration

The base of the Linux configuration is Raspberry Pi OS Lite, a Debian-based image that contains only a minimal operating system. The RPi OS version we use was released on 2024-03-15 [22, 23] and uses version *6.6.20* of the Linux kernel. Since Unikraft RPi only supports a single core at the moment, we set *maxcpus* to 1 in */boot/cmdline.txt*. This makes the comparison between Unikraft and Linux more reasonable.

We compiled the TCP program on the RPi and used it during testing like any other program would be used.

We installed Nginx manually by compiling it from source [5] to make sure its version matches with Unikraft's *lib-nginx*. Since the Unikraft configuration uses an in-memory file system, we placed the test *.txt* files in a ramfs folder.

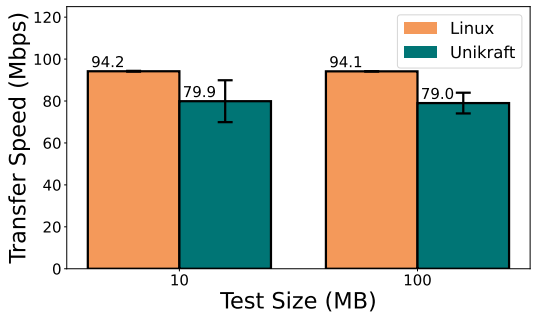


Figure 2: *Average TCP throughput of Unikraft and Linux for different buffer sizes. The error whiskers display the standard deviation.*

5.5 TCP benchmarking

The main goal behind benchmarking the TCP throughput is to gauge the network stack performance. We keep this test simple, using only a single connection. The results of this test help provide some additional context for the Nginx test in the next subsection. We ran the TCP benchmarking with the following two configurations on both platforms:

- 100 test repetitions of 10 MB each
- 50 test repetitions of 100 MB each

The results of the benchmark are displayed in Figure 2. Keep in mind that the hard throughput limit as defined by the hardware is 100 Mbps. We can see Linux reaches 94 Mbps, a bit under the hardware maximum and almost exactly at the expected value considering TCP overheads. Unikraft's throughput is roughly 16% slower, between 79 and 80 Mbps between the two tests. This is an expected result, likely due to the drivers, network stack, and their integration being less optimized than Linux. Figure 2 also shows a high standard deviation for Unikraft, whereas that of Linux is barely visible on the graph. The reason for this might be due to the polling mode netdev. To confirm the causes of this slowdown and standard deviation, further research is needed.

Reporting CPU utilization during this test would provide valuable insights. Due to time constraints and Unikraft not providing a built-in way to collect CPU utilization statistics, we were unable to collect and include this data in this study.

5.6 Nginx benchmarking

Benchmarking Nginx was done using WRK, requesting files from 64 B to 256 kiB in size over 30 seconds. We ran two rounds of tests, one with a single connection and another test with 10 concurrent connections. WRK reuses the same connection to avoid repeatedly including connection setup overhead in measurements. The results of this testing can be found in Figures 3 and 4.

Examining requests per second for the single connection test (Figure 3a), Unikraft outperforms Linux by up to 25% (4 kiB). Above the 16 kiB file size, Unikraft expectedly loses its edge over Linux as the network throughput seemingly becomes the main bottleneck.

Inspecting requests per second for the 10 concurrent connections test (Figure 3b), Unikraft is between 78% and 20% slower than Linux. Unikraft's performance stays very close to what is achieved using a single connection while the performance of Linux scales up significantly. Starting at the 4 kiB file size, Linux's data throughput stagnates and stays around 89 Mbps for all following sizes. Additionally, Unikraft occasionally experiences connection timeouts. This significant performance loss compared to Linux might be connected to how *lib-lwip* handles concurrent connections and threading. However, further research is needed to confirm the real cause of this slowdown.

While running these benchmarks, we came across an issue with Nginx on Unikraft. The request per second throughput of Nginx declines as time goes on. For example, running a similar test with WRK over 5 minutes, Unikraft's average performance comes out significantly lower compared to a 30-second test. Restarting the RPi brings the performance back up again. Unikraft's maintainers confirmed this behavior is an issue observed across its ARM platforms and needs further investigation. Between each 30-second WRK test on Unikraft, we restarted the RPi to reduce the impact of this issue. Performance degradation already starts in this 30-second window, but limiting the test to this timeframe still gives a lower bound for performance gains possible with Unikraft. Additionally, the standard deviations of the Unikraft results are distorted due to this issue.

Figure 4 presents 90th percentile latency data. Examining the latency graph for the single connection test in Figure 4a, we see Unikraft's latency is higher than Linux in most cases but it does stay fairly close. Figure 4b displays the latency data for the 10 connections test where Unikraft's latency is high, between 252 ms and 385 ms for file sizes 64 B through 4 kiB. For those file sizes, Linux's latency is between 1,8 ms and 3,8 ms (only a slight increase over the single connection latency). For files 16 kiB through 256 kiB, Unikraft's latency drops below that of Linux. This might be due to the network becoming the main bottleneck for these file sizes so Unikraft can recover in the latency area. As previously mentioned, Unikraft has problems during the 10 connection test which need to be further investigated to find the definite causes of this latency distribution. Unikraft's latency might improve, especially for the single connection test, if testing is run for substantially longer than 30 seconds but due to the limitations discussed previously we have kept the test a 30 seconds.

All obtained WRK data is available on GitHub [26].

6 DISCUSSION

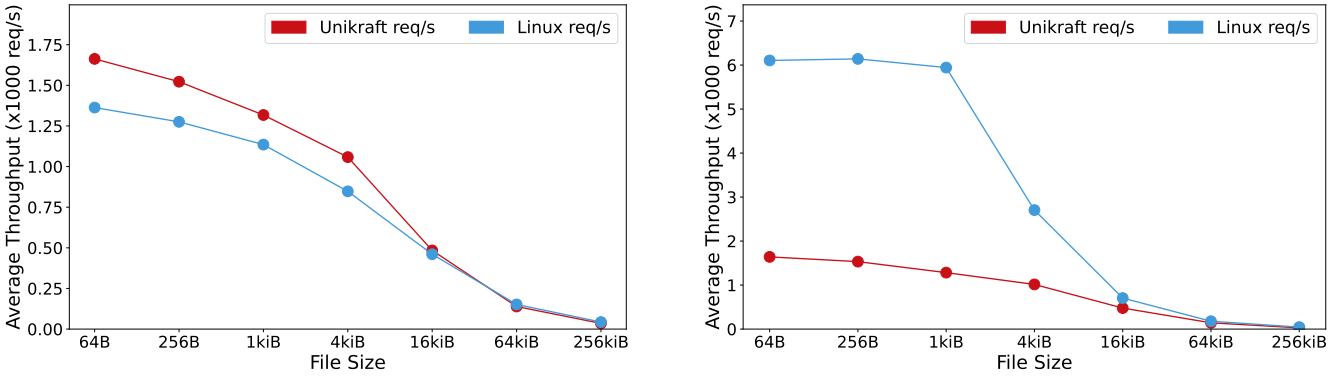
The implementation of the Raspberry Pi platform for Unikraft has demonstrated promising results, but also revealed several limitations and areas for future improvement that call for further discussion.

Due to time constraints, some major limitations are still present in the RPi platform library. The first and most important limitation is the fact that the platform does not support multi-core processing. Secondly, the RPi 3B contains the PrimeCell UART module (PL011) [21] and Unikraft has support built-in for this module. However, the current platform re-implements its own serial console instead of reusing this common code. Ideally, the interrupt system code currently present in the platform should be replaced by Unikraft's implementation for the ARM Generic Interrupt Controller (GIC) interface. Using this common GIC code would most likely reduce the work involved in supporting multi-core processing for the platform.

Using the Unikraft implementations for PL011 and GIC is limited by the fact that no device tree is loaded. Both Unikraft's common PL011 and GIC implementations require the device tree. Having a usable device tree would also make it easier to support different drivers for RPi devices. These limitations should be the main focus of future updates to the platform as this would unlock large performance gains, reduce code duplication, and improve the integration of Unikraft and the RPi platform library.

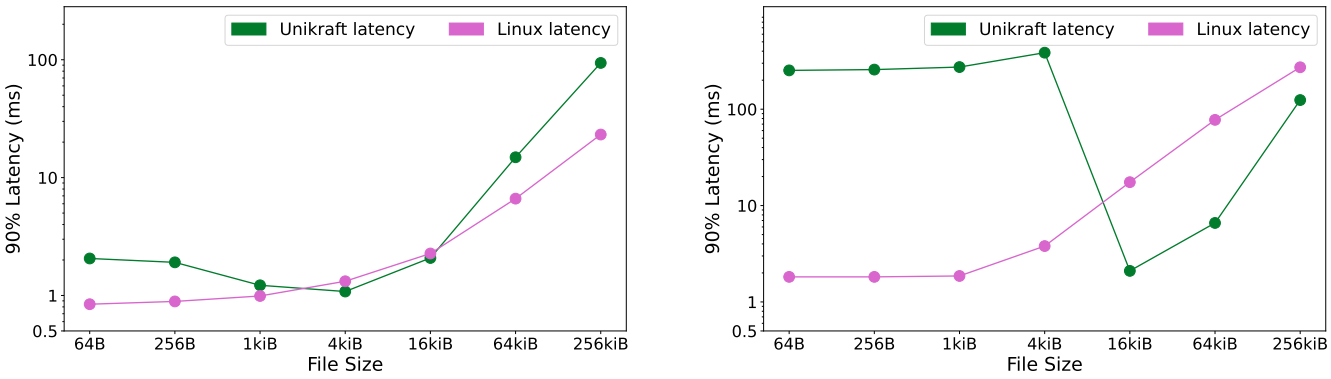
Unikraft on embedded systems is still in an early phase. Device support is limited and even if effort is put into expanding platform support, some limitations hurt Unikraft's attractiveness. One major example of this is 32-bit support. Unikraft used to support 32-bit systems but this has been removed with no plans of bringing it back. Many embedded systems run on 32-bit architectures, making it so Unikraft cannot be taken into consideration when creating software for these devices.

The RPi platform currently has limited driver support. The RPi platform updated for this thesis also supports the RPi 3 B+ and it would be fairly easy to support additional RPi devices. Despite this,



(a) *Benchmark with 1 concurrent connection.* **(b)** *Benchmark with 10 concurrent connections.*

Figure 3: ***Average Nginx requests per second of Unikraft and Linux across different file sizes, measured using WRK.***



(a) *Benchmark with 1 concurrent connection.* **(b)** *Benchmark with 10 concurrent connections.*

Figure 4: ***Nginx 90th percentile request latency for Unikraft and Linux across different file sizes, measured using WRK. Note that the y-axis is presented on a logarithmic scale.***

the network driver we developed is specifically for the RPi 3B and each additional RPi device would need its own network driver.

A *driver-shim* layer has been proposed and tested to improve driver support [13]. The goal of this *driver-shim* layer is to reduce platform porting times and to improve access to various drivers by running unmodified drivers from FreeRTOS [2]. FreeRTOS does not support RPi devices so an alternative solution needs to be found to bring *driver-shim* support to the RPi.

To uncover the specific bottlenecks that cause reduced performance in the situations we covered during the evaluation, further research is needed. Additionally, benchmarking a larger array of different applications would provide further valuable insights into the state and limitations of the RPi platform.

7 CONCLUSION

In summary, Unikraft on RPi, ARM, and embedded systems holds potential for networked applications. We demonstrated potential performance gains by benchmarking an updated version of the RPi platform. While our results revealed a peek into these gains, we also identified lacking areas where performance falls behind Linux. From these results, we identify the following three main areas where the RPi platform needs improvement:

(1) **Multiple connection handling:** Enhancing the RPi platform's ability to manage concurrent network connections efficiently.
(2) **Multi-core support:** Leveraging the full capabilities of the processor to achieve competitive performance when comparing against a more standard Linux setup.
(3) **Driver support:** Expanding the support for various types of hardware to increase the versatility of the RPi platform.

These advancements will make Unikraft a real contender for developers seeking to deploy networked applications on RPi devices.

REFERENCES

[1] [n.d.]. *Contributors to IncludeOS.* https://github.com/includeos/IncludeOS/graphs/contributors?from=2020-03-01&to=2024-03-01&type=c
[2] [n.d.]. *FreeRTOS - Real-time operating system for microcontrollers.* https://freertos.org/
[3] [n.d.]. *Installing NGINX Open Source.* https://docs.nginx.com/nginx/admin-guide/installing-nginx/installing-nginx-open-source/

[4] [n.d.]. *musl libc.* https://musl.libc.org/
[5] 2018. *Nginx version 1.15.6 download.* https://nginx.org/download/nginx-1.15.6.tar.gz
[6] 2024. *Internet of Things (IoT) Market.* https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307
[7] The Unikraft Authors. [n.d.]. *The fast, secure and open-source Unikernel Development Kit.* https://github.com/unikraft/unikraft
[8] The Unikraft Authors. [n.d.]. *Nginx configuration files.* https://github.com/unikraft/app-nginx/tree/staging/rootfs/nginx/conf
[9] The Unikraft Authors. [n.d.]. *Unikraft is a fast, secure and open-source Unikernel Development Kit.* https://unikraft.org/
[10] The Unikraft Authors. [n.d.]. *Unikraft lwIP Library.* https://github.com/unikraft/lib-lwip
[11] The Unikraft Authors. [n.d.]. *Unikraft lwIP Library.* https://github.com/unikraft/lib-lwip
[12] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom).* https://doi.org/10.1109/CloudCom.2015.89
[13] Daniela Andreea Dumitrache. 2021. *Unikraft driver-shim library.* https://github.com/unikraft/driver-shim
[14] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science* (2001).
[15] Inc F5. [n.d.]. *Nginx.* https://www.f5.com/go/product/welcome-to-nginx
[16] Will Glozer. 2021. *wrk - a HTTP benchmarking tool.* https://github.com/wg/wrk/tree/master
[17] Unikraft GmbH. [n.d.]. *The Compute Platform for Lightspeed Services.* https://kraft.cloud/
[18] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14).* USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity
[19] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21).* ACM. https://doi.org/10.1145/3447786.3456248
[20] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, Joel Nider, Mike Rapoport, and Costin Lupu. 2019. Unleashing the power of unikernels with unikraft. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19).* Association for Computing Machinery. https://doi.org/10.1145/3319647.3325856
[21] ARM Limited. 2007. *PrimeCell UART (PL011) Technical Reference Manual.* https://developer.arm.com/documentation/ddi0183/latest/
[22] Raspberry Pi Ltd. [n.d.]. *Raspberry Pi OS Lite ARM64 2024-03-15 files.* https://downloads.raspberrypi.com/raspios_lite_arm64/images/raspios_lite_arm64-2024-03-15/
[23] Raspberry Pi Ltd. [n.d.]. *Raspberry Pi OS Lite ARM64 Release Notes.* https://downloads.raspberrypi.com/raspios_lite_arm64/release_notes.txt
[24] Raspberry Pi Ltd. 2016. *Raspberry Pi Firmware Wiki - Accessing Mailboxes.* https://github.com/raspberrypi/firmware/wiki/Accessing-mailboxes
[25] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *Queue* (dec 2013). https://doi.org/10.1145/2557963.2566628
[26] Job Paardekooper. [n.d.]. *Artifacts, including experiment results and graphs.* https://github.com/jobpaardekooper/unikraft-rpi/tree/measurements
[27] perbu. 2018. *We're porting IncludeOS to ARM64.* https://www.includeos.org/blog/2018/port-to-arm.html
[28] Sharan Santhanam, Simon Kuenzer, Hugo Lefeuvre, Felipe Huici, Alexander Jung, Santiago Pagani, George-Cristian Muraru, Stefano Stabellini, Justin He, and Jonathan Beri. 2020. Towards Highly Specialized, POSIX -compliant Software Stacks with Unikraft: Work-in-Progress. In *2020 International Conference on Embedded Software (EMSOFT).* https://doi.org/10.1109/EMSOFT51651.2020.9244044
[29] Rene Stange. [n.d.]. *Circle.* https://github.com/rsta2/circle
[30] Rene Stange. [n.d.]. *USPi.* https://github.com/rsta2/uspi/tree/master
[31] Rene Stange. [n.d.]. *USPi Ethernet Sample.* https://github.com/rsta2/uspi/tree/master/sample/ethernet
[32] Lynx Software Technologies. 2022. *lynx software technologies releases the industry's first commercial unikernel with posix compatibility.* https://www.lynx.com/press-releases/lynx-software-technologies-releases-the-industrys-first-commercial-unikernel-with-posix-compatibility
[33] Lynx Software Technologies. 2022. *LynxElement: the industry's first unikernel.* https://www.lynx.com/products/lynxelement

ARTIFACTS

Based on the standard Artifact Appendix template used by the USENIX Security Symposium.

A ARTIFACT APPENDIX

This appendix explains how to set up and run the benchmarking discussed in this thesis. The repository containing all code and results can be found at [26]. We will now provide a walk-through of how to set up and benchmark the TCP and Nginx applications on both Unikraft and Linux.

A.1 Abstract

The main artifact is the Unikraft Raspberry Pi 3B platform. The *measurements* branch of the repository contains additional artifacts such as scripts for simplifying the setup and for generating graphs.

A.2 Description & Requirements

A.2.1 How to access. The repository containing all code and results can be found at [26].

A.2.2 Hardware dependencies. The following hardware is needed to reproduce the benchmarks discussed in this thesis:

(1) A Raspberry Pi *3 Model B V1.2* connected via ethernet supporting at least 100 Mbps with preferably 1 Gbps capacity.
(2) A host machine to run the server side of the TCP test and the WRK side of the Nginx test. Any modern machine will most likely be ok since it should easily be able to keep up with the performance of the RPi device. For the specific results discussed in this paper a 16-inch 2021 MacBook with M1 Pro chip and 32 GB of memory was used.
(3) An ethernet connection between the main host and the RPi supporting at least 100 Mbps with preferably 1 Gbps capacity. The connection between the two devices should preferably be as simple as possible. For the specific results discussed in this thesis we connected these devices using a Cat5e ethernet cable combined with a 1 Gbps USB-C to ethernet adapter on the MacBook side. We enabled the connection through the macOS internet sharing feature with the MacBook connected to a wifi network.
(4) A power supply for the RPi of precisely 5 V and 2500 mA.
(5) A monitor, USB keyboard and HDMI cable to interact with Linux running on the RPi.
(6) A micro SD card with at least 16 GB of space. We specifically used a Sandisk Ultra A1 64 GB micro SD card.
(7) Optionally, a UART cable to view some of the console output of the RPi when running the Unikraft tests.

A.2.3 Software dependencies. The following software is needed to reproduce the benchmarks discussed in this thesis:

- A docker container running Ubuntu *24.04* was used to compile the Unikraft images.
- WRK *4.2.0* [16] is required for testing Nginx. In our specific case we compiled it on the MacBook using Apple clang version *15.0.0 (clang-1500.1.0.2.5)* and the *-O3* optimization flag.
- The Raspberry Pi Imager tool to install Linux onto the micro SD card.
- A copy of the Raspberry Pi OS Lite image [22] version *2024-03-15*.

A.3 setup

A.3.1 Installation. First, flash the RPi OS Lite image onto the SD card using the RPi imager tool. Update the *config.txt* file in the *boot* partition of the SD card to include the following options:

```
enable_uart=1
core_freq_min=500
arm_64bit=1
boot_delay=0
disable_splash=1
```

Now insert the SD card into the RPi, connect the monitor, keyboard, ethernet and finally the power cable. Clone the GitHub repository onto the RPi using the following command:

```
git clone https://github.com/jobpaardekooper/unikraft-rpi.git
git checkout measurements
```

Download the Nginx *1.15.6* source from [5]. Build and install from source using default options by using the instructions at [3].

Update the Nginx configuration to the settings found at [8]. Do not forget to restart the service. Now create a ramfs folder using the following command:

```
sudo mount ramfs -t ramfs /ram-dir
```

Place the folder found at unikraft-rpi/measurements/test-files inside the ramfs folder. Update the Nginx configuration to serve these files.

Compile the TCP client program located at:

```
unikraft-rpi/measurements/download-client/main.c
```

using clang with the *-O3* optimization flag. Make sure to update the IP at line 57 to the IP address of the main host machine. Compile another version of the *main.c* file with the settings at the top updated to the following:

```
#define BUFFER_SIZE 1000000
#define TIMES_TO_SEND_BUFFER 100
#define REPEAT_TEST 50
```

Now shut down the RPi and update the *cmdline.txt* file in the *boot* partition of the SD card to include *maxcpus=1* option.

Now on the main host machine also clone the GitHub repository as described above and checkout the *measurements* branch.

Compile the TCP server program located at:

```
unikraft-rpi/measurements/download-server/main.c
```

using clang with the *-O3* optimization flag. Compile another version of the *main.c* file with the settings at the top updated to the following:

```
#define BUFFER_SIZE 1000000
#define TIMES_TO_DOWNLOAD_BUFFER 100
#define REPEAT_TEST 50
```

Now we can prepare the 3 Unikernel images needed to compare against Linux. This was done inside the Ubuntu docker container on the main host machine. Make sure to clone the GitHub repository as described above and checkout the *measurements* branch. Now run the following commands:

```
cd measurements/build/
./build-nginx.sh
```

In the *build* folder where you currently are you will get a *kernel8.img* when this script finishes. Store this file as this is the Nginx unikernel we will use later to benchmark Unikraft.

Now run the following command to generate an image for the 10 MB TCP test:

```
./build-tcp.sh 50 <IPv4 address of the main host machine>
```

Again, save the output *kernel8.img* for later.

Now run the following command to generate an image for the 100 MB TCP test:

```
./build-tcp.sh 100 <main host IPv4 address>
```

Again, save the output *kernel8.img* for later.

A.3.2 Basic Test. Boot the RPi with the Linux installation. Now run *lscpu* and confirm only a single core is online. Keep in mind that every time you reboot the RPi Linux installation you need to re-create the ramfs directory as described previously.

Run the command below to confirm that you can download the test files from Nginx:

```
wget http://<RPi IPv4 address>/test-files/64.txt
```

To test the TCP applications you can start the server program on the main host machine and after that start the client program on the RPi. You will see a log message on the host machine telling you that a connection was established.

To test the Unikraft versions of the TCP or Nginx applications, create a backup of the Linux *kernel8.img* file in the *boot* partition of the SD card and replace it with the *kernel8.img* of the application you want to test. You can test if the application is working correctly in the same way as described above for Linux.

A.4 Evaluation workflow

A.4.1 Major Claims.

(C1): Unikraft RPi achieves 16% lower maximum TCP throughput than Linux. This is proven by (E1), described in 5.1, 5.2, 5.3, and 5.4, whose results are reported in 5.5 and Figure 2.

(C2): Unikraft RPi achieves up to a 25% improvement in Nginx requests per second compared to Linux for a single concurrent connection. This is proven by (E2), described in 5.1, 5.2, 5.3, and 5.4, whose results are reported in 5.6 and Figure 3a.

(C2): Unikraft RPi achieves 20%-78% less Nginx requests per second compared to Linux for 10 concurrent connections. This is proven by (E3), described in 5.1, 5.2, 5.3, and 5.4, whose results are reported in 5.6 and Figure 3b.

A.4.2 Experiments.

(E1): *[TCP benchmark] [1 human-hour + 1 compute-hour]: In this experiment, we will run the TCP benchmarks and evaluate the outputs.*
Preparation: We need the TCP testing applications and Unikernel images created in section A.3.
Execution: Start the server program on the main host machine and then start the corresponding (10/100 MB test) client program on the RPi. After each test the server program will log an array with numbers and you will have to restart the program to perform the next test. There should be 4 total tests to do: 10 MB Unikraft, 100 MB Unikraft, 10 MB Linux, and 100 MB Linux. Make sure to use the matching server program for each client program that you test.
Results: All the resulting number arrays printed by the server should be put in their corresponding *.txt* file in the *measurements/results/<platform>* folder. The 10 MB Unikraft test should go into *unikraft/tcp-100.txt* and the 100 MB test into *unikraft/tcp-50.txt*. The results of the Linux test should go into those same files in the *linux* folder. Now set the working directory to the *measurements/results* folder and run the evaluation script using:

```
python3 tcp.py
```

This should log data to the terminal that looks roughly as follows:

```
Linux 10MB: Avg: 94.19 Mbps, Std Dev: 0.19 Mbps
Unikraft 10MB: Avg: 79.91 Mbps, Std Dev: 9.99 Mbps
Linux 100MB: Avg: 94.15 Mbps, Std Dev: 0.01 Mbps
Unikraft 100MB: Avg: 79.02 Mbps, Std Dev: 4.94 Mbps

Unikraft is x% slower compared to linux:
10mb: 15%
100mb: 16%
Average: 16%
```

Additionally, a *tcp.pdf* file is created which should roughly match Figure 2.
These results prove the claim made in (C1).

(E2): *[Nginx 1-c] [1 human-hour + 15 compute-minutes]: In this experiment, we will run the Nginx benchmarks using a single concurrent connection and evaluate the outputs.*
Preparation: We need to install WRK and create the Nginx Unikernel and Linux setup as described in section A.3.
Execution: First make sure the Linux installation of Nginx is running and serving the test files as described in section A.3. Now run WRK on the main host machine with the following command:

```
wrk -c 1 -d 30s -t 1 --latency \
http://<RPi IPv4 address>/test-files/<file size>.txt
```

Replace *<file size>* with the numbers 64, 256, 1024, 4096, 16384, 65536, and 262144. After each test is complete, copy the output of WRK into the file *linux/wrk-1.txt* with 2 blank lines between each result.
Now do the same thing for Unikraft but make sure to restart the RPi each time you start a new WRK test. This is to mitigate the issue described in 5.6. Place the results of each test into the file *unikraft/wrk-1.txt* with 2 blank lines between each result.
Results: With all the outputs placed in their corresponding *.txt* files, we can evaluate the outputs using:

```
python3 nginx.py
```

This should log data to the terminal where the first half looks roughly as follows:

```
Unikraft speed increase (or decrease) for
```

```
1 connection(s) compared to Linux:
64: 22%
256: 19%
1024: 16%
4096: 25%
16384: 5%
65536: -8%
262144: -19%
```

Additionally, the files *nginx-reqs-1.pdf* and *nginx-latency-1.pdf* are created which should roughly match Figure 3a and 4a.
These results prove the claim made in (C2).

(E3): *[Nginx 10-c] [30 human-minutes + 15 compute-minutes]: In this experiment, we will run the Nginx benchmarks using 10 concurrent connections and evaluate the outputs.*
Preparation: We need to install WRK and create the Nginx Unikernel and Linux setup as described in section A.3.
Execution: The steps are the same as the execution steps of (E2) with the WRK command updated to the following:

```
wrk -c 10 -d 30s -t 1 --latency \
http://<RPi IPv4 address>/test-files/<file size>.txt
```

And the outputs should be placed in the *wrk-10.txt* file of the corresponding platform folder.
Results: With all the outputs placed in their corresponding *.txt* files, we can evaluate the outputs using:

```
python3 nginx.py
```

This should log data to the terminal where the second half looks roughly as follows:

```
Unikraft speed increase (or decrease) for
10 connection(s) compared to Linux:
64: -73%
256: -75%
1024: -78%
4096: -62%
16384: -32%
65536: -20%
262144: -38%
```

Additionally, the files *nginx-reqs-10.pdf* and *nginx-latency-10.pdf* are created which should roughly match Figure 3b and 4b.
These results prove the claim made in (C3).

A.5 Notes on Reusability

The Unikraft RPi platform should be able to run on the RPi 3B+ and RPi 4 if you remove the network driver from the platform as this is only supported by the RPi 3B.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.