DEGREE PROJECT

Computation offloading of 5G devices at the Edge using WebAssembly

Gustav Hansson

Computer Science and Engineering, master's level

2021

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Computation offloading of 5G devices at the Edge using WebAssembly

Gustav Hansson

Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

Supervisors:

Olov Schelén, Daniel Olsson, Johan Kristiansson

Abstract

With an ever-increasing percentage of the human population connected to the internet, the amount of data produced and processed is at an all-time high. Edge Computing has emerged as a paradigm to handle this growth and, combined with 5G, enables complex time-sensitive applications running on resource-restricted devices.

This master thesis investigates the use of WebAssembly in the context of computational offloading at the Edge. The focus is on utilizing WebAssembly to move computational heavy parts of a system from an end device to an Edge Server. An objective is to improve program performance by reducing the execution time and energy consumption on the end device.

A proof-of-concept offloading system is developed to research this. The system is evaluated on three different use cases; calculating Fibonacci numbers, matrix multiplication, and image recognition. Each use case is tested on a Raspberry Pi 3 and Pi 4 comparing execution of the WebAssembly module both locally and offloaded. Each test will also run natively on both the server and the end device to provide some baseline for comparison.

Contents

Chapter 1

Introduction

1.1 Background

With a steadily growing amount of the world population connected to the internet [1] and a rapid increase of Internet of Things (IoT) devices, the amount of data flowing through the internet is growing. Some predictions say that saying that it will increase from 33 Zettabytes in 2018 to around 175 Zettabytes by 2025 [2].

One paradigm that has been emerging to support this significant increase in internet traffic is the idea of Edge Computing where data is processed closer to the end-user. This idea would make it possible to have computational resources at the boundary of the networks instead of relying on centralized data centers, thus reducing the data flowing through the core of the internet. With the emergence of 5G, this paradigm has the chance to come alive.[3] 5G makes the promise of a more robust connection compared to its predecessors as well as very low latencies, which theoretical might reach down to 1 ms [4]. With the support of this new infrastructure, Edge Computing can be utilized to create time-critical systems running complex programs on small resource-restricted devices by moving heavy computations from the end device to the Edge Server.

1.2 Motivation

With an ever-growing amount of devices connected to the internet and an increasing demand for heavy computational applications in all areas, from VR/AR applications using Artificial Intelligence to mobile gaming, higher pressure is put on the performance of these devices. This demand for performance has been met with devices getting improved hardware; faster processors, higher amount of memory and bigger batteries but have also led to higher prices. By moving heavy computations from the end device to an Edge Server, a small device has the possibility of running high-performance applications without the need of expensive hardware. These heavy computations also consume a lot of energy. Thus when the computation is moved, the possibility exists of reducing the

energy consumption on the device, reducing the problem of the bottleneck that is the battery.

1.3 Problem Definition

Q1: Can WebAssembly be utilized to decrease execution time on programs when offloading computations from an end device to an Edge Server?
With end devices running tasks, the execution time increase with the size and complexity of the problem. With these end devices being resource-restricted, will the execution time of the task decrease when offloading the heavy computation to an Edge Server? Will the offloaded execution time using WebAssembly be faster than the native execution of the code on the end device?

Q2: When should the end device offload a task to the server?
How much effect does the latency between the device and the server have on the execution time? What impact does the size of the offloaded payload have on the response time?

Q3: Does the energy consumption on the end device decrease by offloading computations to an Edge Server?
Can offloading be used to increase the battery life of mobile devices?

1.4 Delimitations and Scope

The delimitations of this thesis are to focus on computation offloading part of Edge Computing. Other problems such as acquiring the IP address of the Edge Node, network connectivity issues, and movement of the physical end device between nodes are out of this thesis scope. Because the 5G research environment is not available, the experiments will use WiFi instead of 5G. As there are no physical Edge Nodes to use, a virtual machine running in the RISE ICE Datacenter[5] is used to represent the Node. However, the research is still applicable to an Edge node due to work being kept inside the Edge environment's constraints with a low response time between the end device and the Edge Server.

With the focus of this thesis being on how WebAssembly can be utilized for computation offloading, the scope is restricted to only WebAssembly as a container technology. The time spent on other possible container and virtualization technologies has thus been kept to a minimum.

1.5 Methodology

To research these questions in section 1.3, a proof-of-concept system will be developed and evaluated. WebAssembly (WASM)[6] was chosen as a container technology for its portability and the possibility to run it on most types of hardware. Another primary

reason for the choice is that Wasm is not restricted to any one programming language. The evaluation of the system will be based on measuring execution times of the same Wasm module running both on a local client and offloaded to a server. The same WebAssembly runtime will be used both locally and remotely to ensure a comparable result. Furthermore, the same code used to initialize this runtime and parse inputs will run on both locations. Finally, to get a baseline in the comparison, the same code compiled down into WebAssembly will be executed natively on the local client.

1.6 Thesis Structure

This thesis provides some background theory needed to understand the concepts behind offloading to the Edge. It includes the technologies that have come before and what enables this development. Furthermore, it goes through some related studies that examine similar areas before describing the implementation of the researched solution and how the tests have been performed. Then the result is presented, followed by a discussion around the result, ending with some conclusions drawn and future work.

Chapter 2

Theory

2.1 Virtualization

Virtualization of computer hardware is a technology that first saw the light of day in the 1960s to divide the mainframe computers of the time. This division allowed the mainframes to execute multiple programs parallel to each other. [7] This has evolved to the point where one physical computer can run multiple virtual machines (VM) whom each behaves like its own individual system with virtualized hardware and an isolated kernel.

2.1.1 Containerization

With the virtualization of hardware creating a big overhead, a need arose for more lightweight alternatives. With the release of Docker in 2013, containerization has become a widely used technology to solve this issue. A container is an isolated user-space instance where a program can run and execute as it would in a separate physical or virtual computer. The isolation creates a sandbox where the program inside the container cannot access resources outside the sandbox. This isolation reduces the risk of code inside one container affecting other containers or the host system itself. [8]

The container allows for a *Build Once, Run Anywhere* principle with the container bundling up the application code, config files, and other dependencies. The container runtime is then tasked with being a bridge between the container and the operating system. This runtime allows a developer to only develop for the container platform without considering what physical hardware the final program will be deployed on. With the container not virtualizing or emulating any hardware, the code itself would run on the host system's kernel, thus reducing the performance overhead compared to a VM. [8]

2.1.2 Infrastructure as a Service

The rise of virtualization has enabled data centers to host services allowing customers to rent resources there in the form of virtual machines. These virtual computers act like their own individual servers and allow customers to host services without thinking about the infrastructure working in the background. This abstraction leaves the maintenance of the physical servers to the data center operators, and the customer can focus more on the development of their products. When the programmers are developing and deploying their service, they still need to consider aspects like what operating system the program should run on in the virtual machine, what runtime to use, and setting up the environments.[9]

2.1.3 Function as a Service

Function as a Service (FaaS, also known as Serverless Computing) is a service where code hosted in a cloud is executed to respond to events. With FaaS, the VM, operating system and execution of application code are abstracted away from the developer. This abstraction means that the developers can focus all their work on the application code. The best practice with FaaS is to have isolated functions with a limited scope where the function does one thing and one thing only. Then when an application is running on a user device, it can request the specified function when needed. The cloud service provider then handles the routing of the request and the execution of the requested function.[10]

2.2 Edge Computing

Edge Computing is the idea of moving computational resources from the big data centers placed in the network's core out closer towards the end devices. The aim of having computational power at the edge of the network is to reduce the physical distance the data produced by the end-users have to travel before it is processed. The goal is to improve the application's performance and reduce the load on the Internet as a whole.[3]

2.3 Computation Offloading

Computation Offloading is the principle of moving resource-demanding tasks from a local unit to an external processing unit. It is a research area that has grown out of mobile battery-powered devices performing ever more complex tasks. Offloading has mainly been used and researched between end devices and data centers. With the development of 5G and Edge Computing's rise, the focus has shifted from offloading to centralized data centers out towards more geographically distributed Edge Nodes located closer to the user. [11]

2.3.1 Offloading Approaches

There are, in general, two major approaches to offloading. The first is a more fine-grained process where the programmer specifically states which parts should be offloaded to the server. This method makes it possible for fine-tuned offloading where only the parts of the program considered costly are offloaded. The drawback of this approach is that a developer needs to decide what parts of the program should be offloaded, increasing the time spent on the task and subsequently impacting the development costs. The other is a more coarse approach where the actual process, which is often executed in a VM, is migrated from the client to the server. This migration is a slower procedure that takes more time and resources where the current computing process needs to be stopped, moved from the client to the server, and then be booted up again. This approach abstracts the offloading away from the developer removing the impact on development time at the possible cost of higher latencies. [11]

2.3.2 Computation Offloading at the Edge

When offloading to a data center, the number of offloading endpoints are very limited making it a lot easier to have specialized servers running application specific offloading code. This allows the developer of the application to be offloaded to know which servers that the processes can be offloaded to. However when the offloading is moved from the centers to the Edge, a set of problems emerges. The major problem comes with the increase in the number of servers handling offloading. As developers cannot know exactly where in the world their application will be used they cannot predict which Edge Node the application will offload to. This problem causes a few different issues that needs to be considered when creating an application: [11]

- How to locate the Edge Node to get its IP address
- Generic Edge Nodes can have different hardware with different performances and architectures
- An Edge Node and an end device will have different hardware with a high chance of using different architectures
- Multiple different applications might be supported for offloading, thus the Node cannot store all the needed information about all possible applications
- The user might not be in proximity to an Edge Node
- The internet connection might be unreliable

2.4 WebAssembly

First released in 2017, WebAssembly (Wasm) came as a portable compilation target for most popular programming languages intending to provide execution of low-level code

on the web. Wasm is a binary code format that runs in a stack-based virtual machine providing an isolated runtime for the code to execute in. When a developer writes code in their preferred language, that code can compile down into WebAssembly. This compiled Wasm code can then execute in a web browser or a standalone runtime environment. Using these pre-compiled binaries in the browser, the modules perform faster than native Javascript, as is the standard language used on the web and in the browser. Despite WebAssembly's initial focus on the web, it has become an open standard where tools for running it in a standalone environment and embedding Wasm into other applications are in development.[6]

To improve security when executing a Wasm application, the runtime executes the application in a sandboxed environment where the *Principle of Least Privilege* is applied. Thus the Wasm runtime needs to state what permissions it requires, for example, which files it wants to obtain or if it needs a connection to the internet. The host system grants these permissions, giving the application access to these parts of the host. In and of itself, this method does not provide security. A malicious application can still ask for the rights to acquire files it should not have access to. Despite that, it allows the host greater control in what parts of the system the application has and can thus reduce the risks of unlawful behavior. [12]

2.4.1 WebAssembly System Interface

When WebAssembly first was developed, it could only be executed inside a browser using a set of Javascript code for communication between the browser and the Wasm module. This design decision forced the use of a browser to execute any Wasm code reducing the usability of WebAssembly. The first standalone Wasm runtime moved around this issue by emulating a browser to execute the Wasm module. This emulation was considered an inadequate foundation for the WebAssembly ecosystem to stand on. Thus, the WebAssembly System Interface (WASI) was born. WASI is a general-purpose API to enable the utilization of Wasm in use cases beyond the web. It provides a POSIX-like system call interface giving access to file systems, I/O devices, and network connections. This interface provides portable binaries for the compiled Wasm code to be compiled once and run on any system provided a WebAssembly runtime.[12]

Chapter 3

Related Work

Several studies has been made on computation offloading towards the cloud using different models and use cases.[13][14][15][16][17] The common theme in these studies focuses on when to do the offloading based on either execution times or energy consumption. The studies each have different approaches to how the offloading is done. Some let the developer choose which parts are offloaded, and some systems abstract this control away from the developer. The common conclusion these studies are making is that computation offloading to the cloud can decrease execution times of a program and save power on the device that offloads code to a server. The common bottleneck shown by these studies is that an increase in latency between the client and the server has a big impact on the offloading performance.

The issue that appears when comparing these different approaches is that each solution is limited to one or a few programming languages. In [13], [15] and [16] a developer using these systems is limited to develop using the Java Virtual Machine (JVM). In [14] and [17] a similar situation appears with the use of the .NET Common Language Runtime (CLR) and a Javascript runtime respectively. This limit reduces the usability of these systems, and in an Edge Computing scenario, they would be relatively useless. It forces the developers to use a programming language or a technology that is not suited for the task at hand or forces the Edge Nodes to run several different offloading systems simultaneously.

In [18] the authors look at the issue of running heavy Machine Learning (ML) applications on a hardware-limited device by offloading it to an Edge Node. The approach taken is to create a system that can migrate a whole Javascript application from the client to the server when it is time to do the heavy computation and then migrate it back to the client afterward. The results clearly show a performance increase in execution time when offloading compared to running solely on the client. This approach has the same problem as mentioned earlier with forcing the developers to use a specific programming language.

In [19] the author evaluates the use of WebAssembly as a container technology for Edge Computing. He further compares it to Docker containers in the areas: container startup time, CPU performance, memory utilization, and filesystem I/O performance.

The experimental results show that Docker containers outperform Wasm on CPU and I/O performance. On the other hand, Wasm performed better on container startup times and showed lower memory usage. The author then concluded that for most usage, Docker is the best choice, but shorter programs that are often deployed WebAssembly might be a better fit.

In [20] the authors develop and evaluate a proof-of-concept execution model building on a FaaS architecture. The authors provide an overview of the system design and its workflow. The proposed solution uses WebAssembly as a container technology uses a NodeJS platform to execute the Wasm containers using Emscripten as the WebAssembly compiler. The authors compare their solution with the Apache OpenWhisk platform, which is based on Docker Containers.[21] The results from their experiments show that OpenWhisk provided the faster execution times, but the Wasm runtime provided the quicker startup times. They concluded that WebAssembly could be a viable alternative when used in a FaaS architecture. This paper shows promises in the use of Wasm in offloading. However, the focus of the research was on the serverless platform, thus not showing any comparisons between offloaded execution times and local execution times.

As mentioned above, there have been studies made were comparisons between execution times while executing locally or offloaded has been made, both in a Cloud environment as well as at the Edge. However, these studies are restricted to using one or a few programming languages, which prohibits developers from using the offloading system, thus reducing the usefulness of these systems. The use of WebAssembly can circumvent this issue as it supports a wide array of languages and technologies. Wasm is a relatively new technology. While a few studies on its use for offloading to the Edge have been done, there is an apparent lack of comparisons between local and offloaded program execution times. This comparison is what the thesis aims to achieve, to see how WebAssembly can be utilized to create a general system that allows the system developers to use the language most suited for their specific use case.

Chapter 4

Implementation

4.1 Architecture

This section will go through the design decisions, why they were made, and what the final system turned out to be.

4.1.1 Design Decisions

With previous studies[20][19] showing that WebAssembly performs well on startup times but lacks the raw performance of other container technologies like Docker the architecture chosen follows a serverless structure. The main idea behind this choice is to utilize the fast startup and give full transparency to the developers of what parts of the code might offload to an Edge Server. Letting the developer know what and where things might offload has the drawback of increasing the system's complexity. It forces the developer to think of where the offloading is needed in the program and increases the time spent working on it. Albeit this design process might enlarge the workload on the developers, it can also give more control. Some parts of the program might be sensitive, and the developer might not want it to be sent over the Internet. The Wasm module should also execute locally instead of offloaded if there is no Edge Server in proximity to the user, if the Internet connection is lost or for any other reason where a local execution is preferred. To minimize this added complexity, where the Wasm module is executed abstracted away from the developer. Thus the developer will know what might be offloaded to a server but does not need to decide when it is offloaded.

In the use case of Function as a Service at the Cloud, dedicated servers for serving the functions can be hosted, and a client using this service will always know which server to request. Thus the server itself knows which functions that are needed by its clients. When moving the servers to the Edge, this is no longer the case. If many different systems utilized the Edge Node by hosting their dedicated offloading servers on these Edge Nodes would soon run out of resources. A solution to this issue is to have one general FaaS

system running on an Edge Node that can handle the incoming requests from multiple different services that want to offload. When a service sends a request to the server, the remote system reacts by running the associated Wasm module and returns the result.

However, this method creates a new set of issues. If there are many services an Edge Node cannot store all possible Wasm modules and must acquire these from somewhere. There are two solutions to this issue: the first solution is to send the WebAssembly file in the request from the client to the server, and the second is to let the server fetch the Wasm file from a global repository. The benefit of the first method is that it is an isolated approach and does not require any other systems to be up and running. It also ensures that the same Wasm file is executed both locally and offloaded, reducing the risk of bugs caused by accidentally having different versions on the various platforms. It also means that more data needs to be sent over a connection with a high chance of being wireless. Handling more data will also put more strain on the resource-restricted end device.

Using the second approach requires having a separate service running on a server to provide the Wasm files to the Edge Node. When deploying a product, the developers need to upload the Wasm file to this global repository so the Node can fetch it. This repository allows the end device to supply the name of the wanted module so it can be collected by the offloading server and executed on the Edge Node. Compared to the first approach, the upside with this is that the internet connection between the Edge Node and the global repository is most likely to be a wired connection that is more reliable and faster than the connection between the end device and the Node. This global repository can also be used to host additional files a Wasm module might need to use. For example, in the use case of machine learning, the repository can host the pre-trained machine-learning model. The files fetched from this repository can then be cached by storing them in the Node's local file system, allowing for faster access on later requests.

4.1.2 System Overview

The final system design comprises three major parts, the local program running on an end device, i.e. an IoT or mobile device, the offloading server, which runs on the Edge Node, and the global repository, which runs in a data center. See figure 4.1.

The subsystem on the end device consists of the main program a developer creates. This program executes on the end device and has a part that is considered for offloading. This program uses a library that takes care of the offloading decision and the interconnection to the WebAssembly runtime. Consequently, the developer only needs to create the main program and the WebAssembly module and make a function call to the library with the module as a parameter. The how and where the execution of this module is abstracted away, and the library returns the execution results.

The offloading server is reached via a REST API endpoint with the name of the wanted Wasm module as a parameter and possible inputs. This system checks if the needed files are cached on the server. If they are not, then the files are fetched from the global repository. When all files are there, the server initializes the WebAssembly runtime and executes the module. The result is returned as the response to the API

request.

The third subsystem is the global repository which in this architecture is a third party system that hosts the files needed for the offloading server to execute the requested Wasm module.

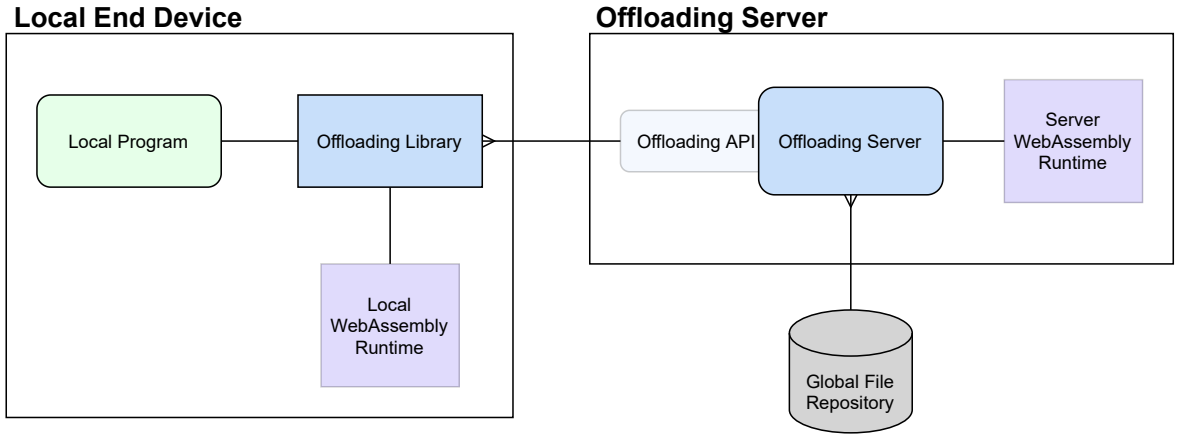

Figure 4.1: Design overview of the offloading system

4.2 Technologies

The implementation of this system is written in Rust as it is a language with a large width of use cases from networking and server backends to embedded IoT devices. Furthermore, it is a fast and efficient language, and it provides memory safety, thus reducing memory violation bugs and vulnerabilities. It also has good support for building a project for a WASI target, making the creation process of the WebAssembly modules easy.

The chosen WebAssembly runtime was Wasmer [22] because it provides support for multiple architectures as well as integration with the most popular programming languages. It is also one of the few Wasm runtimes that have reached a stable release, and it contains a well-developed set of documentation and examples. The primary issue with Wasmer is that it does not support 32-bit ARM CPUs as of writing this thesis.

This combination of language and Wasm runtime allows for the use of the same technologies on both the end device and the Edge Node. Using the same software stacks on both platforms improve the comparability of the result, reducing the risk of having the execution time being affected by differences in implementation.

For the global repository Harbor[23] is used. It is an open-source registry system that allows for storing and fetching WebAssembly modules as well as possible additional files. The service is hosted in the RISE ICE Datacenter.

4.3 Experimental Setup

The setup for each test consists of an end device, a server, and the application code. Each test was done in 3 scenarios:

- On the end device using native Rust
- On the end device using WebAssembly
- Offloaded to the Server

To get a baseline to compare against each test was also executed using native Rust on the offloading server.

These tests compare the native Rust code and the same code but compiled down to WebAssembly. This comparison is made to see what overhead WebAssembly has and what performance is lost by using it. As Wasm is a relatively new technology, it has vast possibilities to get more optimized, so the comparison provides a lower bound for the test. For a use case with a browser environment, this comparison is not very relevant due to a browser not being able to execute native Rust code. For an IoT environment, however, no browser is used, thus indicating if WebAssembly is relevant to use or if it is more beneficial not to use the offloading system at all.

As WebAssembly emerged, it was a compliment to JavaScript to provide faster performances in the browser. As mentioned in section 2.4 WebAssemblys scope moved away from the web. With time it has become a more open standard capable of being used in various situations. This thesis omits comparing the WebAssembly module's performance to JavaScript due to the result not telling anything about the actual overhead of Wasm. If JavaScript and WebAssembly were compared, two different implementations would be done, one in a language compiled to WebAssembly and one in JavaScript. Thus the code executed would not come from the same source providing a high risk of differences in implementations due to language differences, thus reducing the comparability of the result. In this thesis, the tests compare the native code and the same code compiled to WebAssembly. The actual implementation is the same, thus omitting the source of error implementation differences produce.

End Devices

Each scenario was executed on a Raspberry Pi 3 Model B and a Raspberry Pi 4 Model B as end devices to compare the performances with different hardware restrictions. The specifications for the devices can be seen in table 4.1. These two devices are comparable to the two main types of devices that will be used in the 5G network, IoT devices, and mobile phones. IoT devices are usually very small, cheap, and have minimal resources which are comparable to the Raspberry Pi 3 Model B. Modern mobile phones are usually more expensive and a bit more rich in resources with more memory and more processing power and is in these tests represented by the Raspberry Pi 4 Model B.

	Pi3 Model B	Pi4 Model B
Processor	Broadcom BCM2837A1(B0), Quad-core Cortex-A53 64-bit SoC @ 1.2GHz	Broadcom 2711, Quad-core Cortex- A72 64-bit SoC @ 1.5GHz
Memory	1GB LPDDR2 SDRAM	4GB LPDDR4 SDRAM
WIFI Module	2.4GHz IEEE 802.11.b/g/n/ac wireless LAN	2.4GHz / 5.0GHz IEEE 802.11.b/g/n/ac wireless LAN

Table 4.1: Hardware specifications for the 2 end devices. [24]

Edge Server

As there was no physical Edge Node to use, the offloading server was hosted in the RISE ICE Datacenter as a virtual machine. For comparison, two different servers with different virtual hardware were used. No other programs were running parallel on the offloading server. The two servers were both running the same operating system and using the same program code. They are named *Small Server* and *Big Server* in this thesis to distinguish them from each other. The specifications for the servers can be seen in table 4.2.

	Small Server	Big Server
vCPUs	6	24
Memory	8GB	128 GB
Operating System	Ubuntu 20.04 LTS	Ubuntu 20.04 LTS

Table 4.2: Virtual hardware specifications for the 2 virtual servers.

4.3.1 Tests

The tests consisted of 3 different use cases:

- Calculating the *nth* number in the Fibonacci sequence
- Multiplying 2 $n \cdot n$ matrices
- Running an image recognition neural network on an image of size $n \cdot n$ pixels

The Fibonacci test was chosen because it is a very computational heavy task with a low amount of data that needs to be processed. The matrix test is then done to provide a test that handles a large amount of information. These two tests are good benchmarks, but they are not very realistic in real-world use cases. To cover this area as well, the image recognition test was chosen as it can be used by a real-world system.

Execution Time

For measuring the execution time in all three scenarios, each test was executed in 50 iterations with increasing complexity. Increasing complexity meaning an increase in the value of n for each test, i.e., calculating a higher number in the Fibonacci sequence, multiplying bigger matrices and running image recognition on a bigger image. Each iteration was running ten times, with the outputted result being the average of these ten runs. To see where the system bottlenecks are and where the time was spent measurements were done on:

- Transfer time to server - how long it takes to send the request with all the data from the end device to the server
- Fetching files from Harbor - how long it takes to fetch the needed files from the global repository
- Initialize WASI environment - how long it takes to initialize the environment that the module will run in
- Run module - how long it takes to execute the Wasm module

Power Usage

To measure the power usage, the image recognition tests were run on the Raspberry Pi 3 Model B with an image of size $4000 \cdot 4000$ pixels on all three scenarios. This test is limited to image recognition as it is the most relevant test for a real-world application. Each test ran for 24 hours, during which the power usage was measured using a Shelly Plug S [25]. As the execution time on the image recognition for the three scenarios differ, the test consists of a loop that spawns a new thread every 90 seconds. During this time, all three scenarios will be able to finish the previous run, and thus the result will not be affected by a faster scenario running more loops. It simulates a more real-world situation where an IoT device lays dormant until an event happens. This example imitates a surveillance camera that connects to a movement sensor and an image recognition system. As the interesting part of this system in this thesis is the image recognition and the possible performance benefit when offloaded, the camera is abstracted away, and the same static image is sent in every run. For comparison, the Pi's power consumption in an idle state as well as using 100% of its CPU were measured for 24 hours as well.

Chapter 5

Results

5.1 Calculating Fibonacci Numbers

This section will present and analyse the results for the Fibonacci tests.

5.1.1 Raspberry Pi 3 Model B

In figure 5.1 for $n \leq 35$, the execution of Wasm locally has a clear lower time compared to the offloading due to the offloading overhead. At $n = 36$ the curve crosses the offloading curves, and the local Wasm curve grows rapidly away from the two others. As the baseline comparison, the native Rust scenario on the Pi runs without extra overhead, thus have a fast execution time. Its curve then follows the local Wasm curve and crosses the offloading at $n = 38$.

In figure 5.2 we can see the execution time when offloading to the *Big Server* divided into the four primary bottlenecks of the system mentioned in section 4.3.1. The graph shows that up to $n = 35$, the actual time to run the module is a minimal part of the total time. Most is being spent on fetching the Wasm file from Harbor and initializing the WASI environment. The fetching of files and initializing the environment can also be very similar in size and do not grow with n increasing. The graph also shows that the transfer time to the server is a tiny part of the total time. For higher values of n, the vast majority of the time is spent executing the Wasm module.

5.1.2 Raspberry Pi 4 Model B

The graphs for the Raspberry Pi 4 in figure 5.3 looks very similar to figure 5.1 with the difference of the graphs in 5.3 starting to grow a bit later with the local Wasm execution crossing the offloading at around $n = 38$. The local Wasm and the native Rust executions are also a bit lower than their Raspberry Pi 3 counterparts.

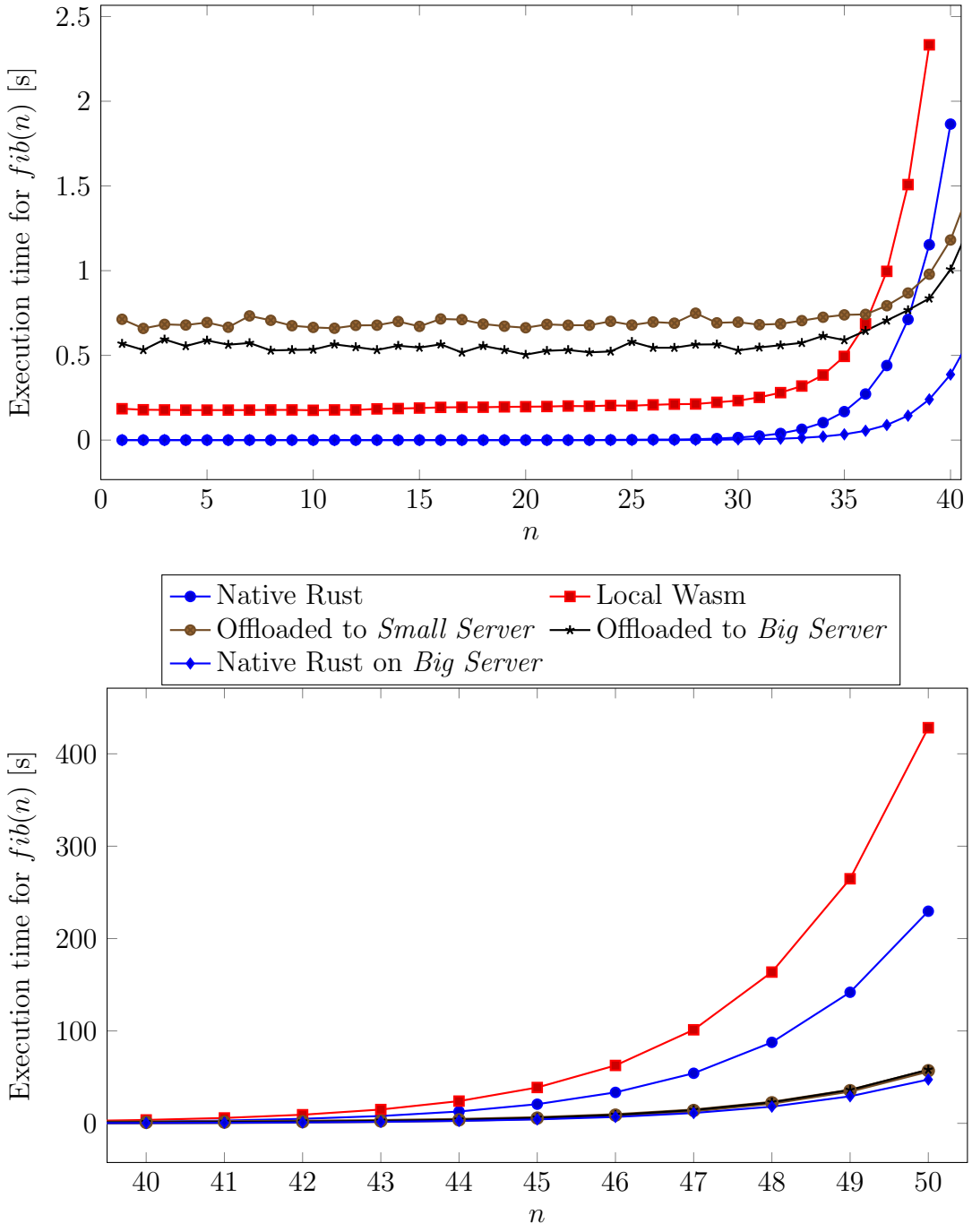

Figure 5.1: Total execution time for calculating the nth fibonacci number on the Raspberry Pi 3 Model B

In figure 5.4 the server-side operations are generally identical to their respective parts in 5.2 as they are executed in the same environment. The transfer time to the offloading server is a bit lower in the Pi 4 tests compared to the Pi 3 ones.

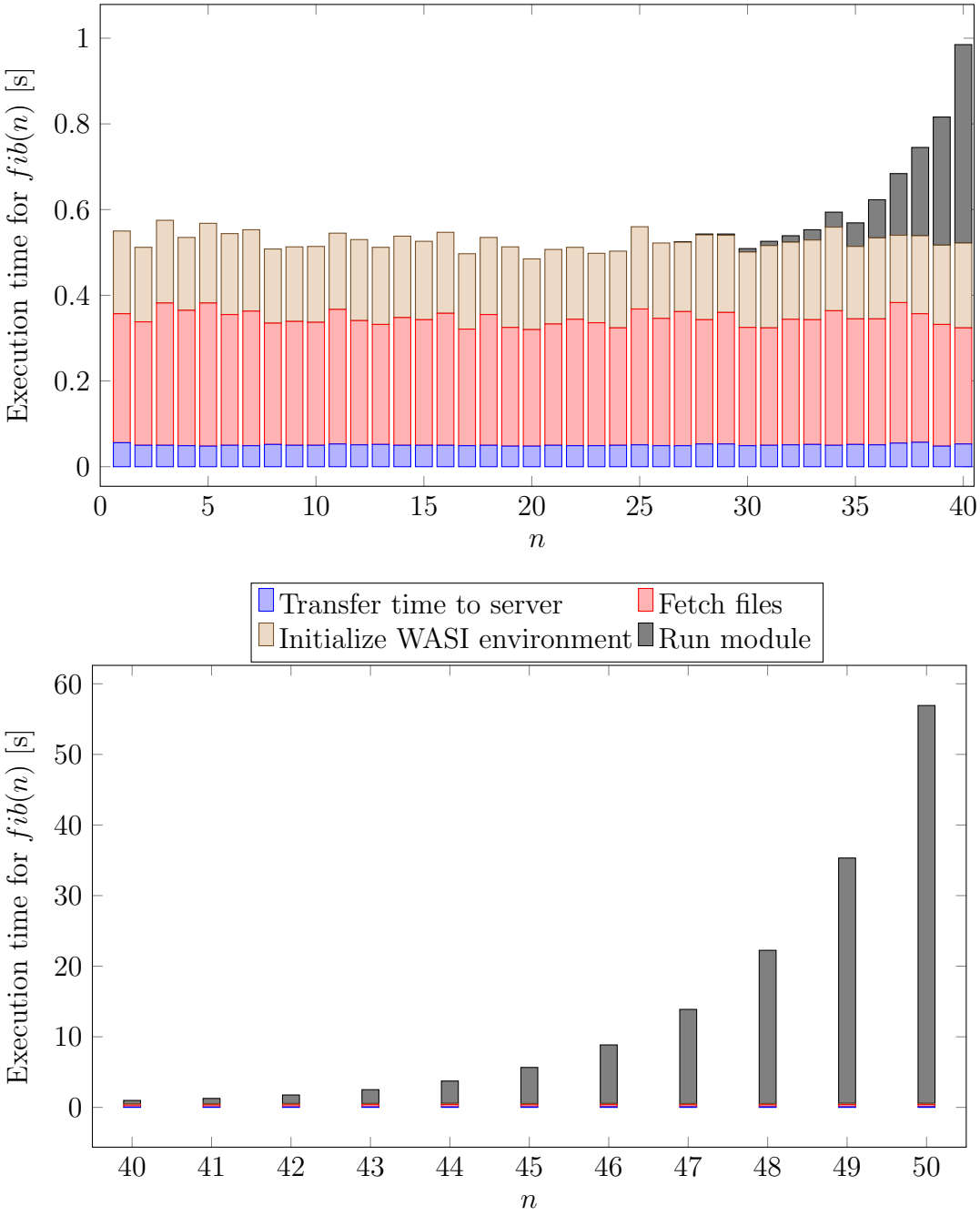

Figure 5.2: Execution time for calculating the nth fibonacci number offloaded from the Raspberry Pi 3 Model B showcasing where the time is spent.

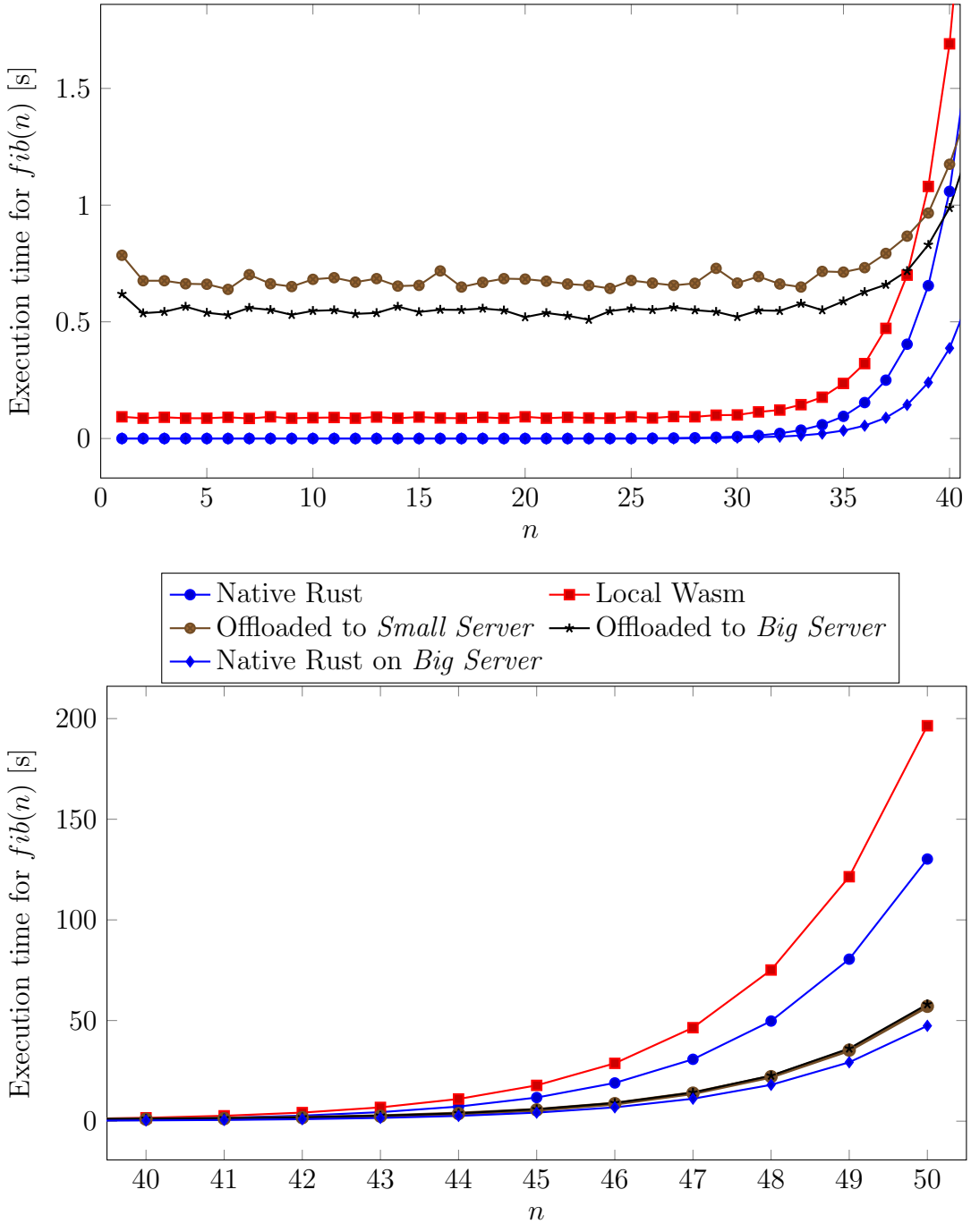

Figure 5.3: Total execution time for calculating the nth fibonacci number on the Raspberry Pi 4 Model B

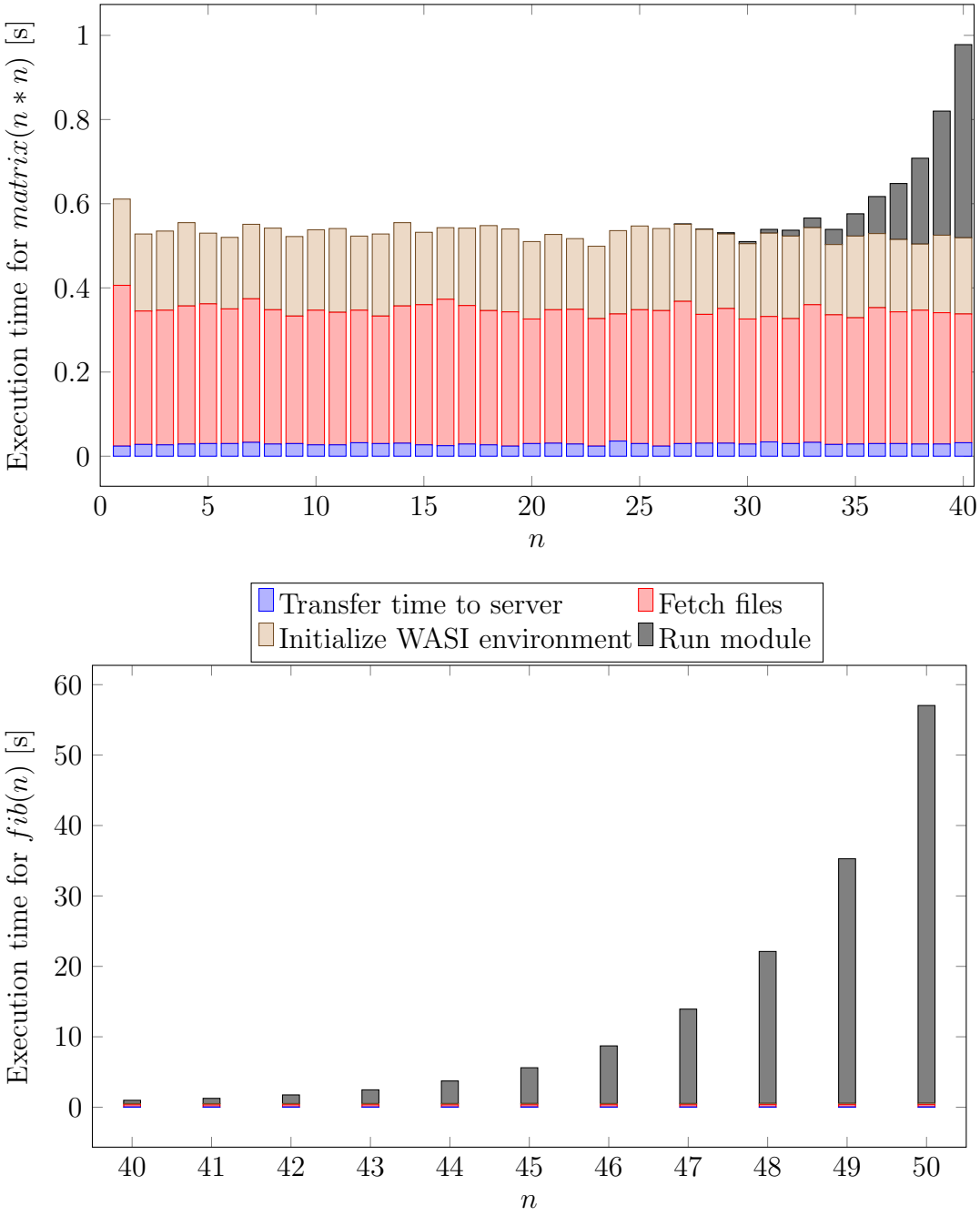

Figure 5.4: Execution time for calculating the nth fibonacci number offloaded from the Raspberry Pi 4 Model B showcasing where the time is spent.

5.2 Matrix Multiplication

This section will present and analyse the results for the matrix multiplication tests.

5.2.1 Raspberry 3 Model B

In figure 5.5 a plot for the matrix size can be seen displaying the relationship between the growth in size with the increase in execution time. For $n \leq 1500$, the local Wasm curve rapidly grows away from the other scenarios, and already at $n = 1500$, the execution time of the local Wasm is higher than the execution time of the other scenarios the largest measured matrix size of $n = 2500$. The three other scenarios stick close together, with the native Rust execution slowly rising away from the offloading ones in the end. In the second graph of figure 5.5, the local Wasm curve is omitted due to its exponential growth away from the other scenarios. At $n = 1500$, its total time is around ten times as much as the other curves and only continues to grow, preventing the comparison of the much closer curves of the other scenarios.

In figure 5.6 the transfer time to the server is a significant part of the total time as opposed to figure 5.2. It is also increasing with the growth of data that is sent to the server. The initialization of the Wasi environment can be seen to be a bit higher than in the 5.2 figure. In the end, higher values of n, running the Wasm module take up the majority of the total time, but with the higher amount of data sent to the server, the transfer time effect is significant.

5.2.2 Raspberry Pi 4 Model B

The plots in figure 5.7 look very similar to the ones in figure 5.5 with the differences of the curves starting to grow at a bit higher values of n and that the native Rust execution being the clear faster scenario. Similar to figure 5.5 the second graph of 5.7 omits the local Wasm curve due to its high exponential growth compared to the other scenarios.

In figure 5.8 we get a similar result as in 5.6 with the difference being that the transfer time to the server is less in the Pi 4. This lowering is most likely due to the Pi 4 being faster at transferring data via WiFi compared to the Pi 3.

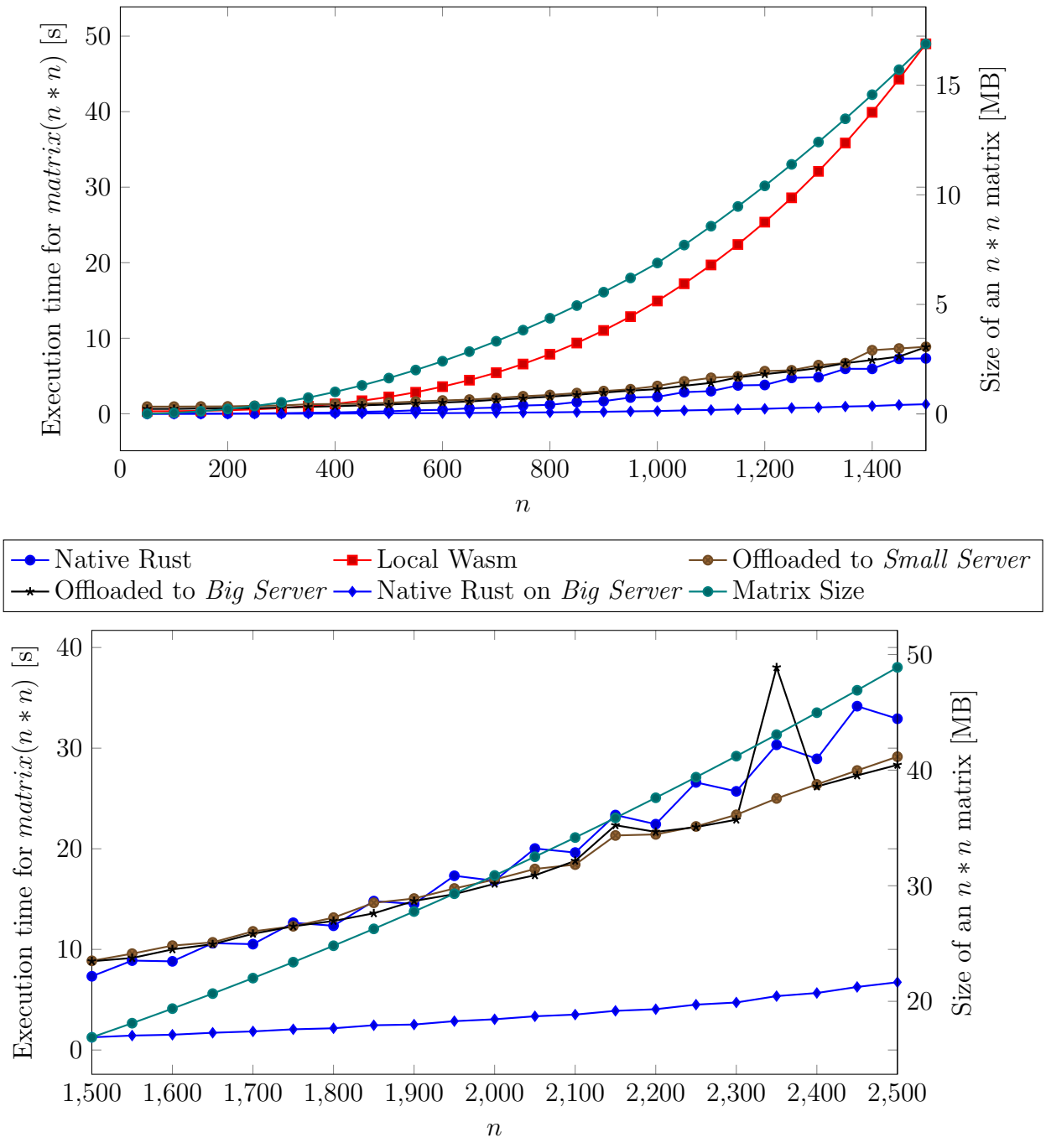

Figure 5.5: Total execution time for multiplying two matrices of size $n * n$ on the Raspberry Pi 3 Model B

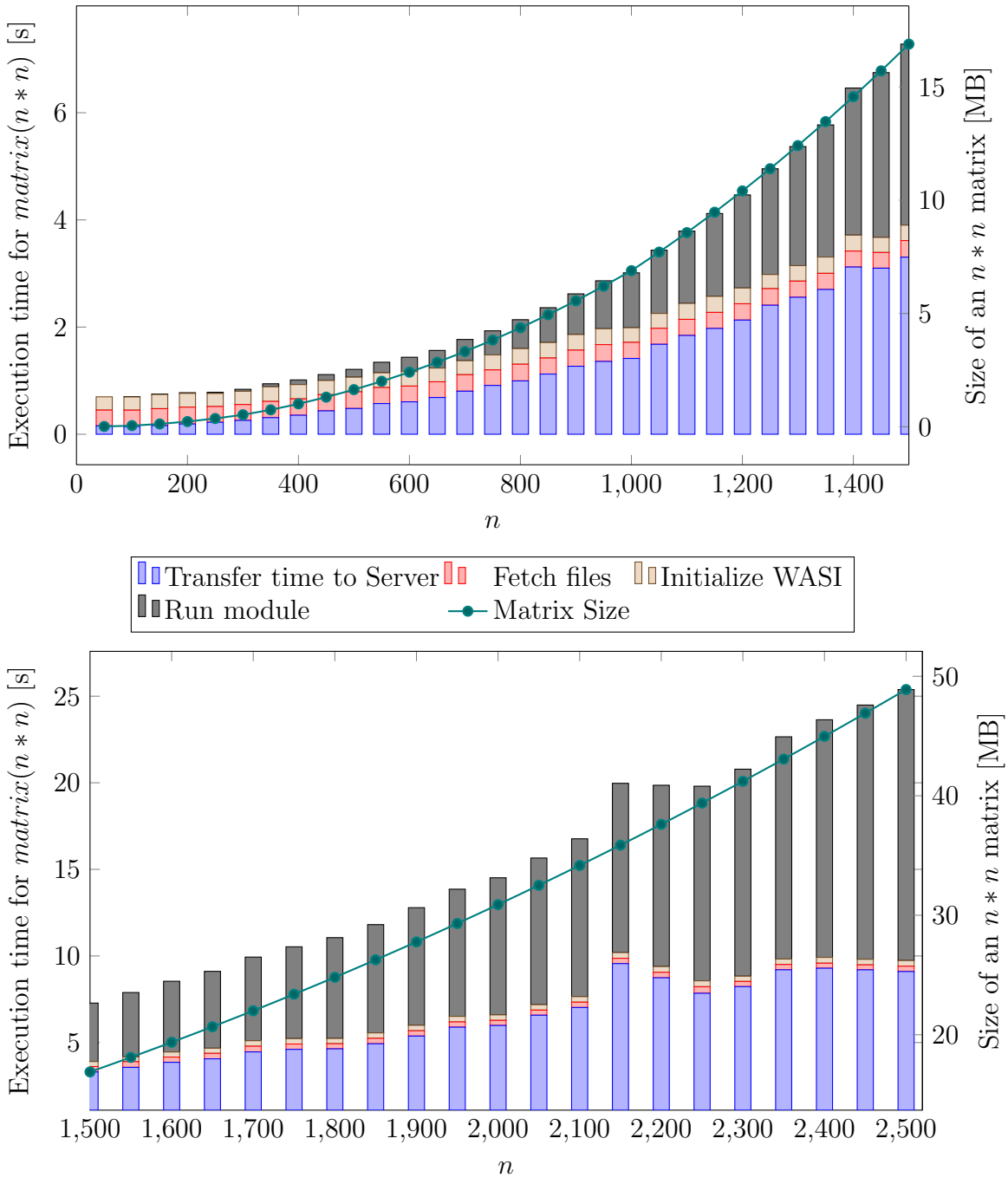

Figure 5.6: Execution time for multiplying two matrices of size n∗n offloaded from the Raspberry Pi 3 Model B showcasing where the time is spent.

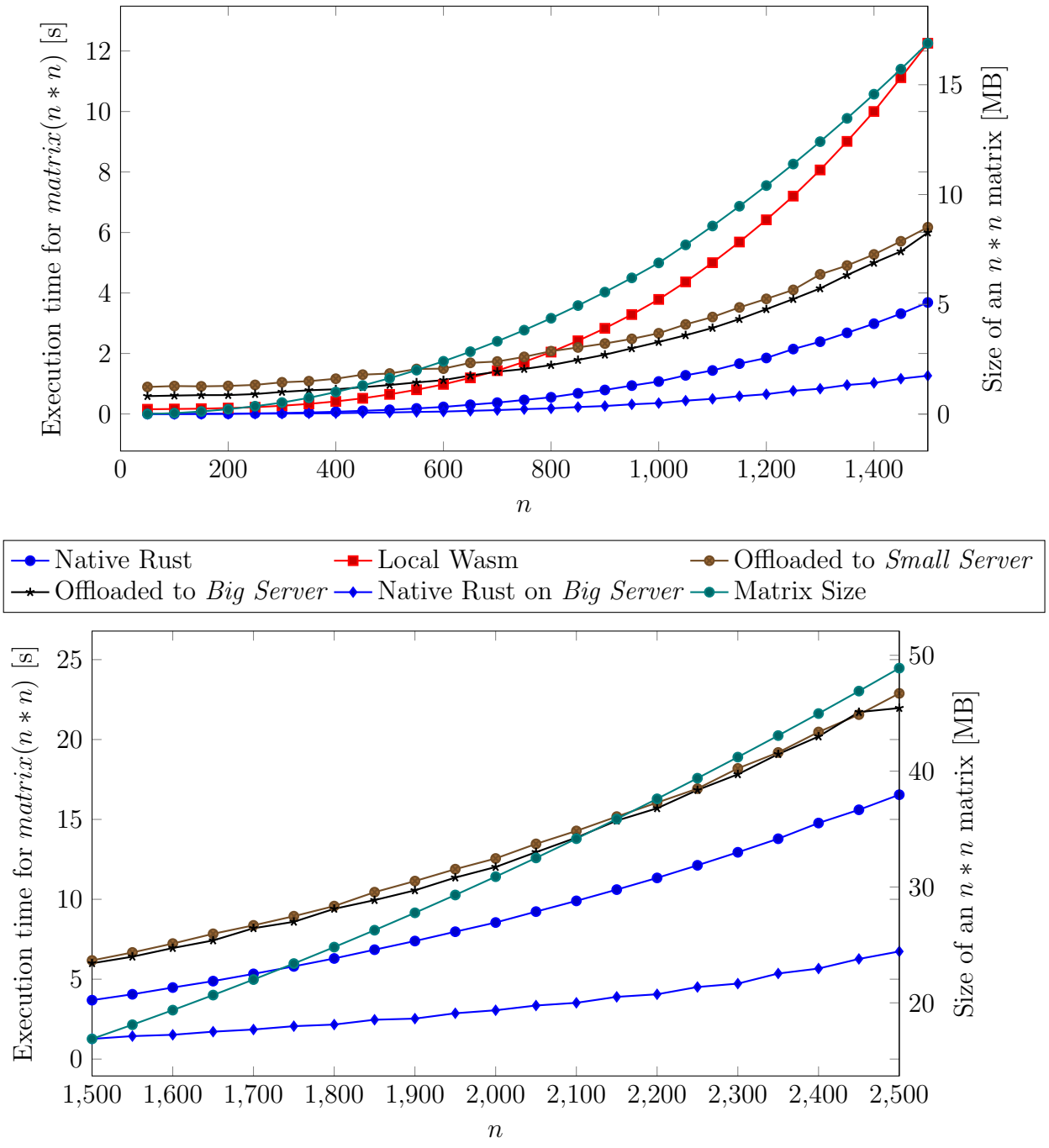

Figure 5.7: Total execution time for multiplying two matrices of size $n * n$ on the Raspberry Pi 4 Model B

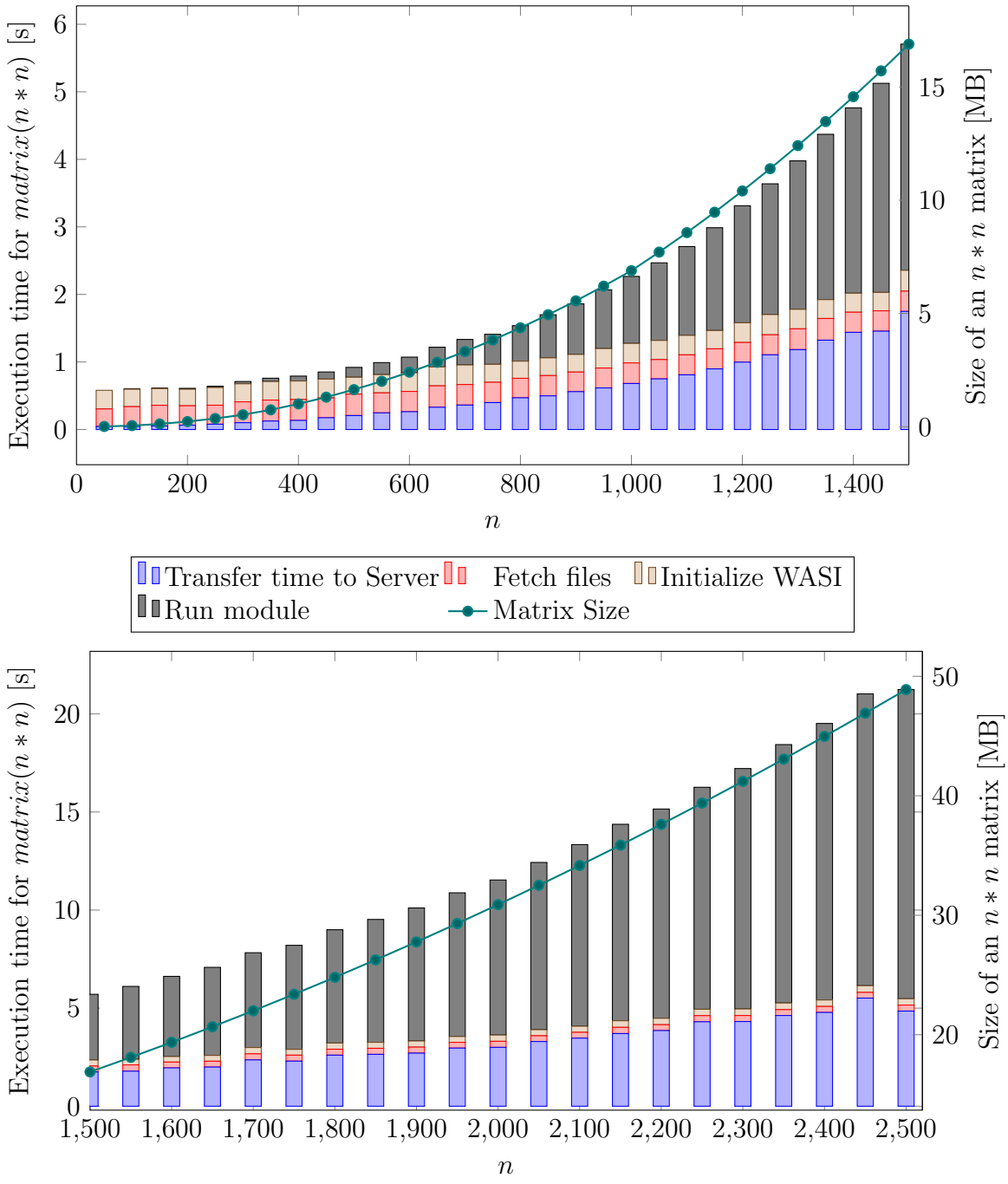

Figure 5.8: Execution time for multiplying two matrices of size n∗n offloaded from the Raspberry Pi 4 Model B showcasing where the time is spent.

5.3 Image Recognition

This section will present and analyse the results gathered by the image recognition tests.

5.3.1 Raspberry 3 Model B

Figure 5.9 shows the local Wasm starting with a total time above 40 seconds. The other scenarios in this test keep together and end with execution times of around 16 seconds. The native Rust execution with its lower overhead keeps under the two offloading scenarios until about $n \approx 9000$ pixels. There it rises above the *Big Server* offloading and approaches the *Small Server* where it in the end just barely moves above the *Small Server* offloading. In the second graph of 5.9, the local Wasm curve is omitted due to being very far away from the other scenarios, which would prevent the comparison of the native Rust scenario with the two offloading ones. Starting at 40 seconds, the local Wasm time is more than two times as big as the slowest of the three other scenarios. Thus, it would only increase the scale of the y-axis and make it harder to compare the other plots.

Figure 5.10 shows that when offloading and sending an image to the server, the transfer time is staying at an insignificant part of the total time, even with the image being over 3 megabytes in size. The execution of the module for smaller image sizes (low values of n) is around 50% of the total time. With the image size increasing, the module's runtime is growing while the fetching of files, server transfer time, and the environment's initialization is kept constant. Thus for high values of n, the vast majority of the total time is spent running the module.

5.3.2 Raspberry Pi 4 Model B

Figure 5.11 shows more linear increases in execution time with none of the curves of the scenarios crossing each other. For example, the local Wasm is starting at around 12 seconds and ends above 30. On the other end, the native Rust begins at 1 second and end at about 7 seconds.

The figure 5.12 is extremely similar to figure 5.10 and does not differ in any significant way.

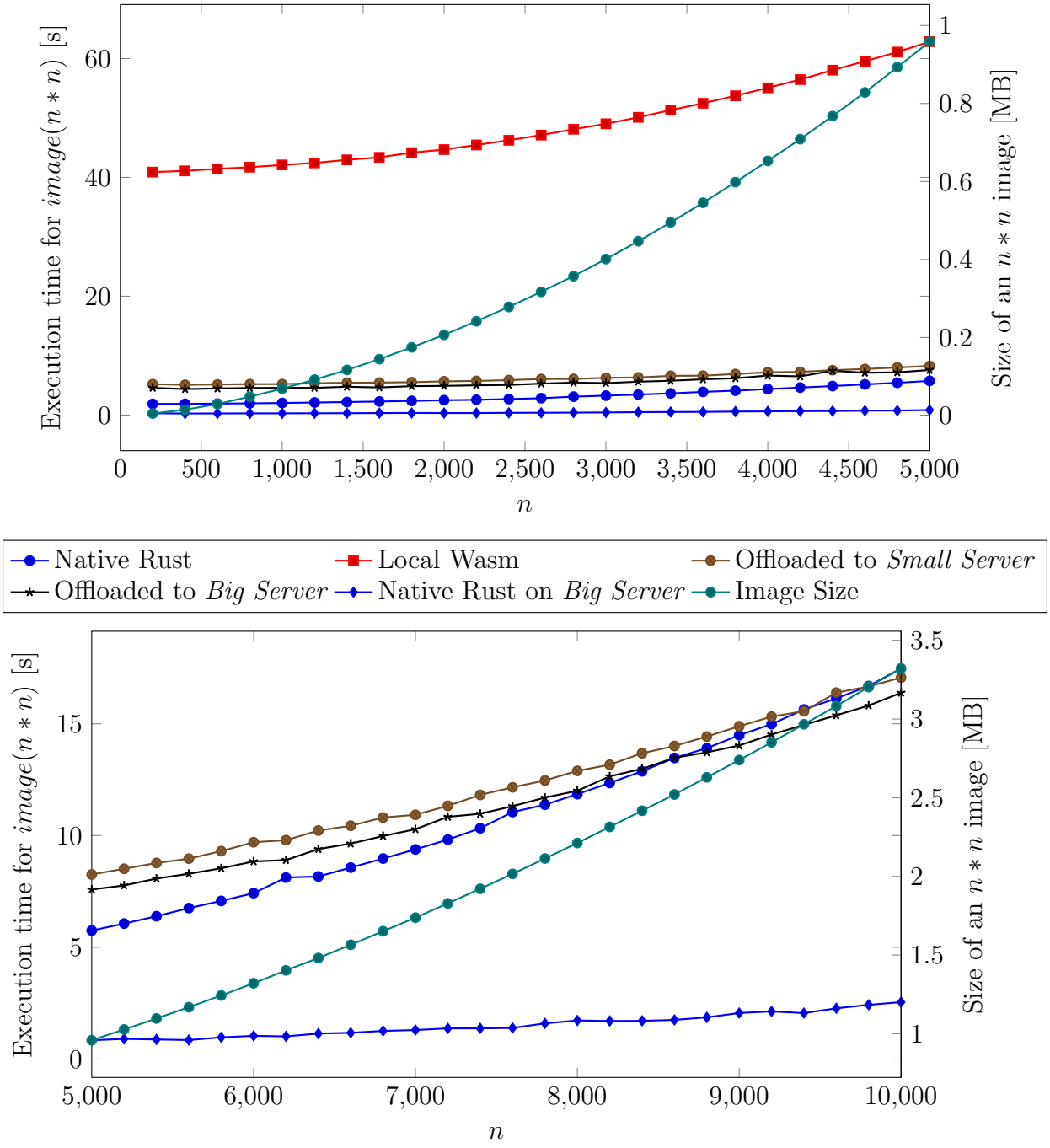

Figure 5.9: Total execution time for running image recognition on an image of size $n * n$ pixels on the Raspberry Pi 3 Model B

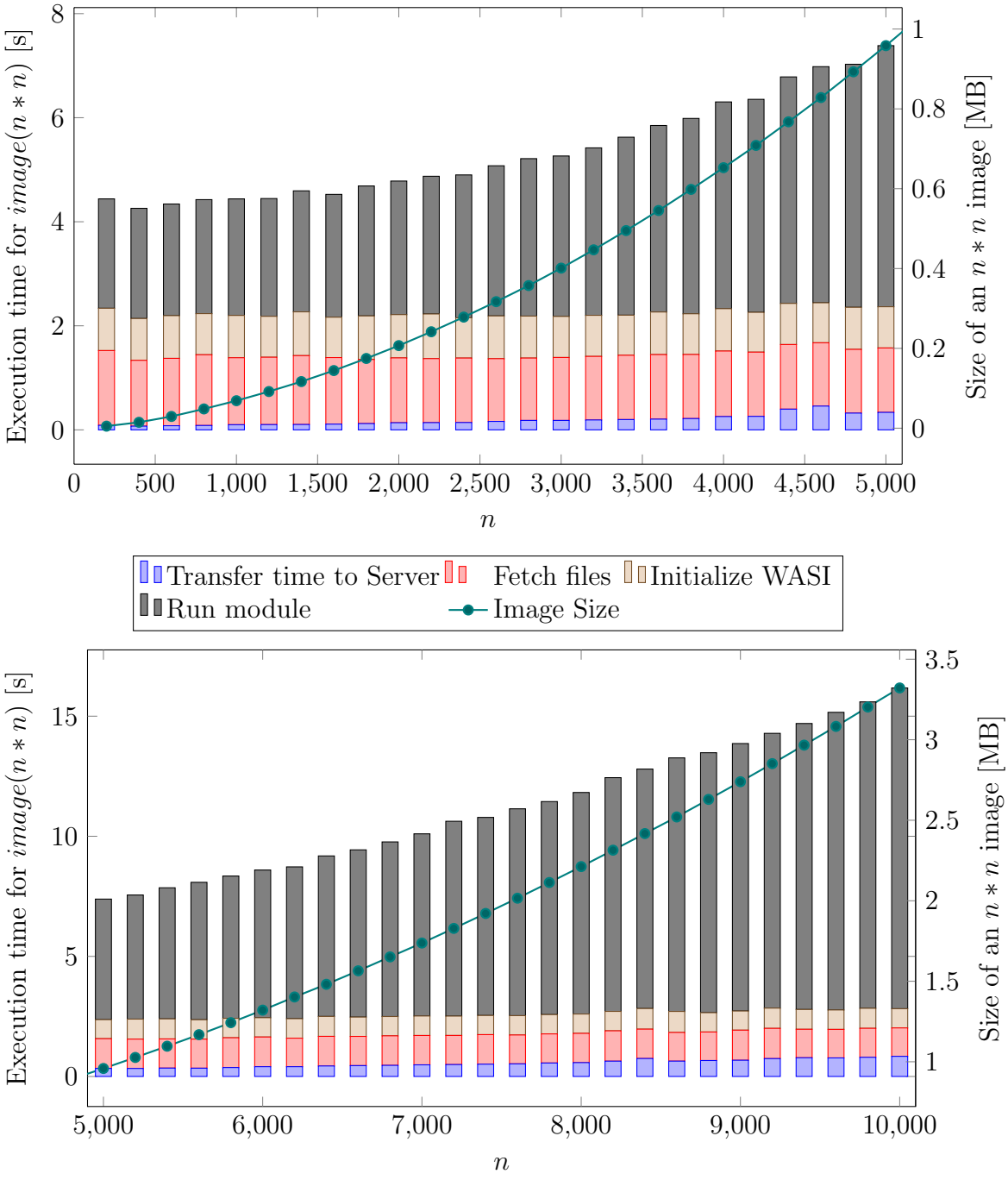

Figure 5.10: Execution time for running image recognition on an image of size $n * n$ pixels offloaded from the Raspberry Pi 3 Model B showcasing where the time is spent.

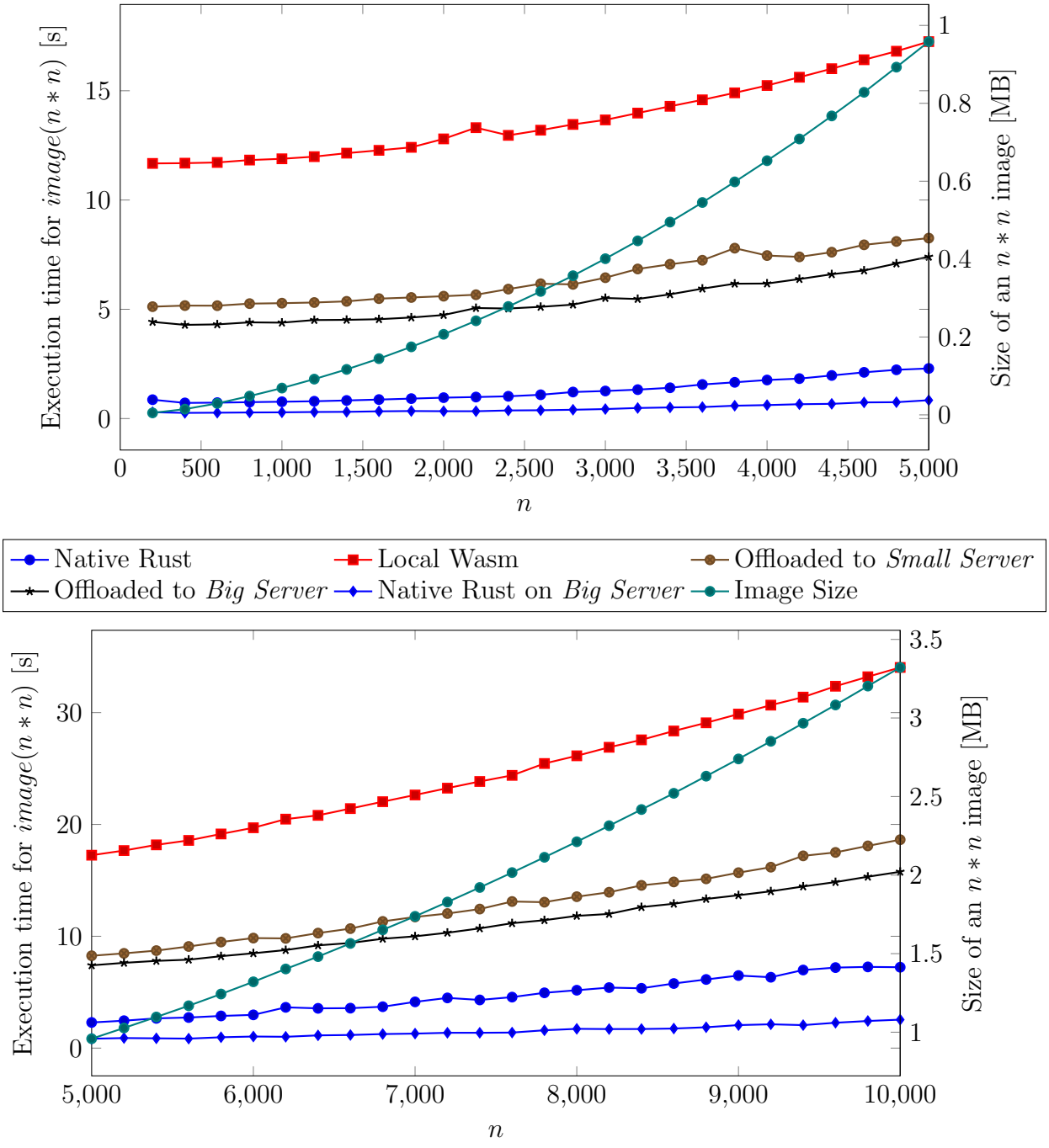

Figure 5.11: Total execution time for running image recognition on an image of size $n*n$ pixels on the Raspberry Pi 4 Model B

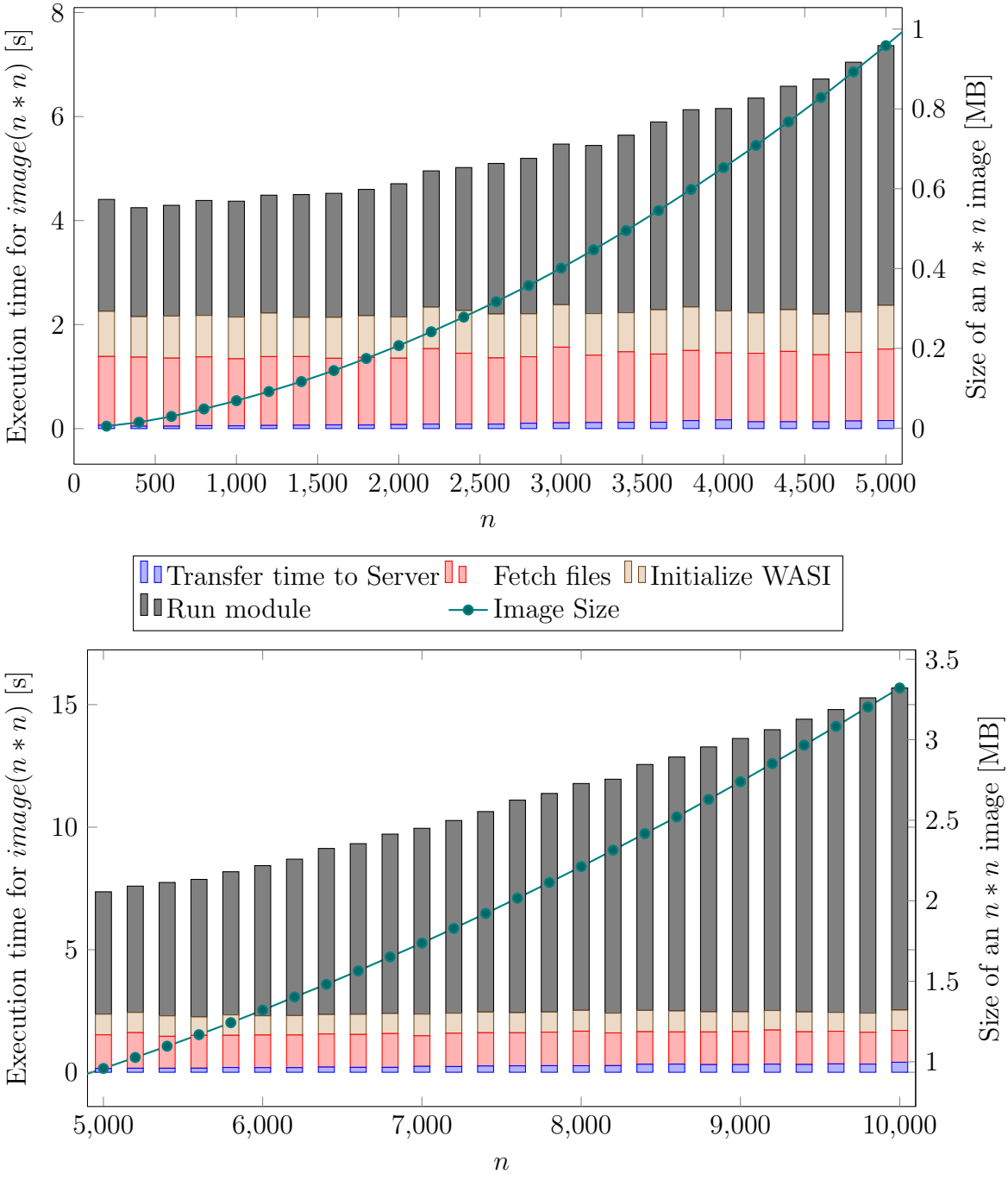

Figure 5.12: Execution time for running image recognition on an image of size $n * n$ pixels offloaded from the Raspberry Pi 4 Model B showcasing where the time is spent.

5.4 Power Consumption

This section will present and analyze the measurement of the power consumption tests. Figure 5.13 displays the result from running the power usage test described in section 4.3.1. The idle state and the maximum CPU scenarios indicate how significant the power consumption is for each executed scenario. As shown in the figure, the offloaded and native Rust examples consume just a little more power than idle. The local Wasm is consuming close to half of what running the CPU on max capacity consumes above the idle baseline.

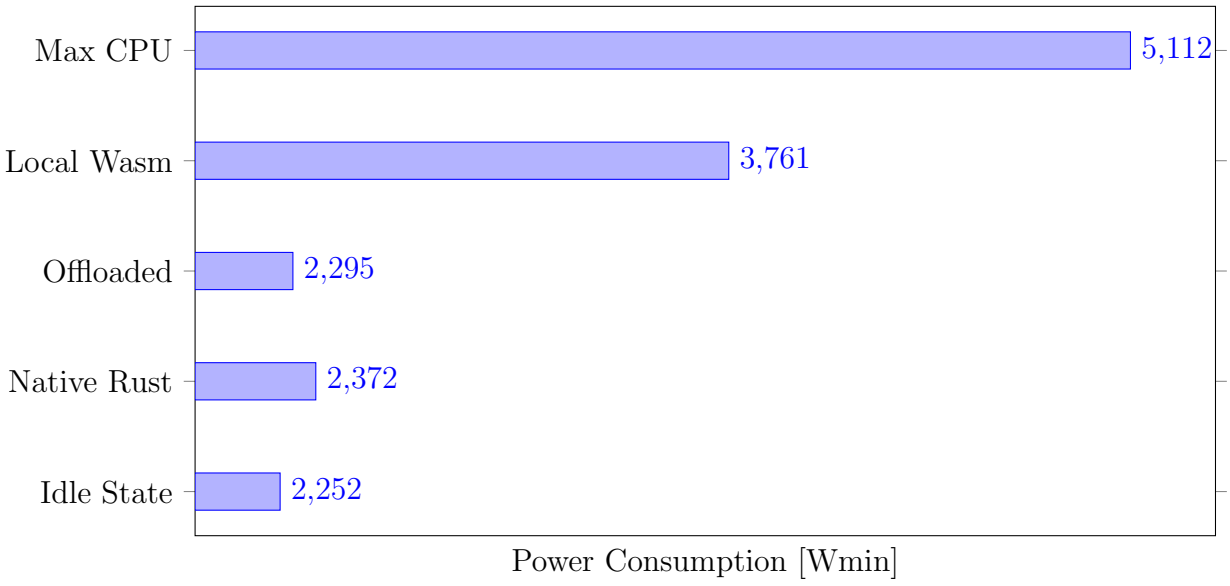

Figure 5.13: Power consumption when running the Raspberry Pi 3 Model B for 24 hours in different scenarios

Chapter 6

Discussion

Figures 5.1 and 5.3 show that with lower values of n, the offloading overhead has a significant effect on the time. This overhead, however, is relatively constant, as can be seen in figure 5.2 and 5.4. With higher values of n, the total time for offloading increases at a lower rate than local execution, wherein the end, offloading is the faster alternative. With the local device having access to less powerful hardware, the curve starts to grow at a smaller n than the offloading server. The curve also rises at a much higher rate. Figures 5.1 and 5.3 show that offloading can even be faster than executing the same code in a native Rust environment for the more complex calculations with high values of n. However, this example is not a very realistic use case as most real-world uses are more than just a heavy computation without any data. Still, it indicates the possibilities that WebAssembly can provide.

Figure 5.5 shows that the local Wasm execution on the Raspberry Pi 3 almost immediately rises rapidly away from the other scenarios. This rise is most likely due to the low amount of memory the device can access. The different scenarios keep close together, with the native Rust being a bit slower than offloading. This test also shows that there are enormous benefits for offloading compared to execution on the device, and there can also be performance gains compared to native Rust. Figure 5.7 shows that with more resources in the Pi 4, the local Wasm stays below the offloading curve for a short while before it shots away. The Pi 4 also shows that locally the native Rust has a big advantage with more computational power than offloading to a server.

In the matrix multiplication, the offloading overhead does not stay constant, as can be seen in figure 5.6 and figure 5.8. This change is explained by the increase in data that is transferred to the server. More data takes more time to move; thus, the time to send data to the server increases with the size of the matrix.

With the image recognition tests being the most realistic in the sense of real-world use, both figure 5.9 and 5.11 give a slow result for executing WebAssembly locally. The result for the Pi 4 (figure 5.11) shows a big difference between native Rust execution and the offloading, with the native being consistently faster. With the less powerful Pi 3 in figure 5.9 the native execution starts with a similar performance benefit as with the

Pi 4. Still, as the image size increases, the difference gets smaller and smaller until the offloading and the native are very similar. The image size seems to have a minor impact on the server transfer time, as can be seen in figures 5.10 and 5.12 and stays a minor part of the total time. Comparing this with offloading the matrix multiplication, where the increased matrix size led to a major part of the time spent sending the data to the server. The matrices' size was an order of magnitude bigger than the image sizes, explaining the big difference.

These tests measuring execution times show that depending on the use case, there are great possibilities with offloading WebAssembly. In an IoT environment where the end device can run the code natively, the performance gain with offloading is negligible. In the use case of the end device running a program in a browser where there is no option of running the native code, the benefits of offloading are huge. For more complex calculations, as in image recognition, the execution time while offloading is at the smallest half of the local comparison. The gap only grows more significant for larger amounts of data.

As can be expected, the performance gain when offloading from the Raspberry Pi 4 is a lot less compared to the less powerful Pi 3. This comparison shows that the smaller the device's resources, the more it can gain by offloading. This result is self-evident as less powerful hardware takes a longer time to execute complex programs, but it indicates the offloading possibilities. It can allow for thin clients containing nearly no hardware to perform as well or even better than a more powerful and expensive device.

As the Pi 4 is relatively rich in resources with a quad-core processor and 4GB of RAM, it is more comparable to a modern mobile phone instead of an IoT device which is more often a very resource-restricted microcontroller. In the use case of a mobile phone, there is no possibility of using native Rust as neither Android nor iOS supports the language natively. For this, the results show substantial performance gains with offloading WebAssembly even though the end device has a relatively high amount of resources.

On the server-side, however, the differences between the two offloading servers are relatively slim. The more powerful server gives a lower execution time, as can be expected, but the performance gain is minimal during these experiments. While the *Big Server* performs a bit better, it also uses a lot more resources than the *Small Server*, 24 virtual CPUs and 128 GB of memory compared to 6 virtual CPUs and 8GB of memory. As these tests are the only load on the offloading server, they do not consider any possibilities for performance loss due to heavy load on the server. When many devices are offloading simultaneously, the more powerful server would most likely have a significant advantage.

With the power consumption displayed in figure 5.13, the consumption from offloading and running the image recognition application natively uses little more power than in an idle state. However, while running WebAssembly locally, the power consumption increases by almost 64% compared to offloaded. This increase is most likely caused by the much longer execution times in the local scenario. Thus, this makes the application running for longer, therefore using the computer's resources for a higher percentage of the measured time. This reduction in power consumption gives a good indication of what

offloading with WebAssembly can do.

With figures 5.2, 5.4, 5.6, 5.8, 5.10 and 5.12 showing that for high values of their respective n the majority of the time is spent on running the Wasm module on the server. Thus the major bottleneck of the system as of now is the actual WebAssembly runtime. As WebAssembly is a relatively new technology, future developments and optimizations can improve the runtime environment and increase execution speeds. Reducing the execution speed is thus a big prospect when it comes to improving the usefulness of offloading.

Other aspects of the system that can have a major impact are the use of 5G as the network technology which can reduce the transfer time to the server. Using 5G have the possibility of reducing the bottleneck that happens with larger datasets that are transferred to the server as seen in 5.6 and 5.8. These tests were done using WiFi due to the lack of access to a 5G network, they do not accurately show how the network affects the offloading but with the promises of faster and reliable 5G connections compared to WiFi, 4G, and older technologies, these conclusions can still be drawn.

The chosen methodology and the measured results give a reasonable indication of the usability of WebAssembly. However, some choices can have harmed the result. The primary aspect is the choice of architecture. In this work, the offloading server starts up a WebAssembly module when it receives a request from the end device. A more practical way to handle it would have been to allocate the resources on the offloading server when the application on the end device starts. Then when the end device makes a request, the WebAssembly environment would already be up and running. Thus removing the time spent on initializing the WASI environment and the time spent fetching the needed files. However, as this would be a more practical approach in a real-world scenario, it would also come with its fair share of issues. For example, how to measure the execution time in a proper way that still utilizes this change. Thus this approach was not chosen, and the results are still giving a good set of answers to the researched questions.

6.1 Sustainability

Utilizing offloading in applications can improve execution times and reduce energy consumption, as seen in this thesis. That means that computational heavy and time-critical systems can use smaller and cheaper end devices. These systems can improve the quality of life for individuals in scenarios like smart cities and self-driving cars. However, there is a cost behind it. Buying the end devices might get cheaper, but they risk being useless without offloading. As the Edge Nodes used can be costly to put up and need maintenance, the price of using offloading might be substantial. In one way, this would pave the way for new businesses and markets. On the other hand, using these Nodes would most likely come with a high cost, driving up the price of using the end devices. Thus, it can bring new jobs to the field while risking further division between the socio-economical classes if society becomes heavily reliant on offloading.

With the main prerequisite behind offloading to the Edge being the Edge Node's geographical distribution, there is another risk of social division. This division can occur

due to the monetary cost of the Nodes, making it inviable for the companies to deploy Nodes in areas with a low population. Thus, the countryside would not have access to the same quality of applications, further incentivizing the ongoing urbanization. These issues apply to all uses of Edge Computing and are not limited to just offloading.

With the offloading having the chance of reducing the power consumption of the end device in the end, it can even reduce the energy footprint of the application. However, the same amount of computations are still performed. They are only moved from the end device to the server. Unlike a mobile phone, a server has more effective cooling and can utilize the excess heat. If the Nodes' excess heat can be used instead of dissipating out in the air, the environmental impact of using mobile phones would be reduced. However, compared to data centers, the centers have an even more significant possibility to utilize this waste heat.

As the Edge Node needs to be close to the end-user, there is a risk that the number of Nodes required would exceed the resources in a centralized data center to cover the land where offloading services are provided. Thus more hardware would need to be used to utilize Edge Computing. With it being harder to use the waste heat from the Node, the environmental impact of Edge Computing has a high chance of being more prominent than the data centers. However, one positive aspect of the much smaller nodes is that they can be placed where they are needed, and with the small size, they can blend in into the background. Compared to a complete data center that needs large areas of land to be built, the Nodes are easier to deploy and require less planning, time, and area to be built.

6.2 Conclusion

This thesis aims to answer the three questions defined in section 1.3.

Q1: Can WebAssembly be utilized to decrease execution time on programs when offloading computations from an end device to an Edge Server?
When offloading the WebAssembly module, the results show that it has significant performance benefits. This gain can especially be seen with more complex computations which handle a large amount of data. For the execution time compared to running the application natively, the performance gain is minimal. Thus, for use cases where the device can run the code natively, it is mostly not beneficial to use WebAssembly. For use cases as in a web application running in a browser, there are substantial possible decreases in execution time with offloading using WebAssembly.

Q2: When should the end device offload a task to the server?
The results show that the more complex the computation, the more significant advantage offloading has. In a situation like calculating Fibonacci numbers where nearly no data is sent to the server, the offloading overhead is substantial for smaller computations. When the complexity increases, the local execution grows significantly faster than offloading. Thus, for complex operations offloading is more suitable. The results also show that the

transfer to the offloading server is often a tiny part of the total time. Even with the transferring time increasing with the growth of data and taking up a more considerable percentage of the whole time, offloading is still mostly more beneficial for more data-demanding tasks.

Q3: Does the energy consumption on the end device decrease by offloading computations to an Edge Server?
Yes, by offloading the WebAssembly module from the end device to the offloading server the end device power consumption is reduced. This reduction could lead to an increase in battery life for a mobile unit. With the offloading reducing the need for powerful hardware on the device, there are also possibilities for the use of less powerful but more energy-efficient components on the device while still having the performance of a more powerful device while at the same time improving the battery life.

6.3 Future Work

With 5G being a significant enabler for practical offloading to the Edge, this is an important field to study. With the 5G research environment not being accessible for use during this thesis, no tests could be done using 5G, and thus it is one of the significant tasks in the future.

The more significant part of the time spent when offloading is to run the WebAssembly module; thus, improving the runtime has the most substantial possibility of enhancing the overall validity of offloading. This thesis uses Wasmer as the WebAssembly runtime due to its broad support of different architectures and that it has reached a stable release as explained in section 4.2. With WebAssembly being a relatively new technology, the development of different runtimes is ongoing. Thus, future studies should compare these existing runtimes to evaluate if the system can gain any performance benefits by switching runtime.

To further explore the usefulness of this system, more tests need to be done on the offloading while the server is under heavy load. This would give more insight into how valid offloading is when there are a considerable number of users. In combination with this, a caching system should be developed and examined, i.e. how to cache the WebAssembly modules fetched from the global repository. In this thesis, the files are gathered, used, and then removed from the offloading server to provide a worst-case and equal scenario for each test. In a real-world environment, however, the offloading system can store these files for later use.

As the offloading server was running on a virtual machine, no power consumption measurements could be made. Measuring this is worth looking into in the future when there are dedicated Edge nodes. With the end device reducing its power consumption, a critical study can be done to see if the end device's total power consumption and the Edge Node are increased or decreased by offloading. If the offloading server uses much more power than the end device, the question of the environmental sustainability of offloading arises. Still, with more research in how to utilize excess heat from an Edge Node, the

heat generated by the Node can be used in heating, reducing the Node's environmental impact.

References

[1] The World Bank, "Individuals using the internet (% of population)," 2019, accessed: 2021-02-23. [Online]. Available: https://data.worldbank.org/indicator/IT.NET.USER.ZS

[2] International Data Corporation (IDC), "The digitization of the world, from edge to core," November 2018. [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[3] K. Shaw, "What is edge computing and why it matters," November 2019. [Online]. Available: https://www.networkworld.com/article/3224893/what-is-edge-computing-and-how-it-s-changing-the-network.html

[4] J. Rogerson, "How fast is 5g?" February 2021. [Online]. Available: https://5g.co.uk/guides/how-fast-is-5g/

[5] RICE ICE, "Rice ice datacenter," accessed: 2021-06-03. [Online]. Available: https://www.ri.se/sv/ice-datacenter

[6] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," February 2017. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3282510

[7] C. D. Graziano, "A performance analysis of xen and kvmhypervisors for hosting the xen worlds project," Master's thesis, Iowa State University, 2011. [Online]. Available: https://lib.dr.iastate.edu/etd/12215/

[8] IBM Cloud Education , "What is containerization?" May 2019. [Online]. Available: https://www.ibm.com/cloud/learn/containerization

[9] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (iaas)," *International Journal of Engineering and Information Technology*, vol. 2, no. 1, 2010.

[10] IBM Cloud Education, "What is faas (function-as-a-service)?" July 2019, accessed: 2021-03-17. [Online]. Available: https://www.ibm.com/cloud/learn/faas

[11] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, 2019.

[12] L. Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web," *Mozilla Hacks*, March 2019. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[13] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: Computation offloading as a service for mobile devices," in *MobiHoc '14: Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, vol. 11-14-August-2014. Association for Computing Machinery, 8 2014, pp. 287–296. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2632951.2632958

[14] E. Cuervoy, A. Balasubramanian, D. K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahlx, "Maui: Making smartphones last longer with code offload," in *MobiSys '10: Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM Press, 2010, pp. 49–62. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1814433.1814441

[15] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 945–953.

[16] N. Geoffray, G. Thomas, and B. Folliot, "Transparent and dynamic code offloading for java applications," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, vol. 4276 LNCS - II. Springer Verlag, 10 2006, pp. 1790–1806.

[17] M. Zbierski and P. Makosiej, "Bring the cloud to your mobile: Transparent offloading of html5 web workers," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, vol. 2015-February. IEEE Computer Society, 2 2015, pp. 198–203.

[18] H. J. Jeong, I. Jeong, H. J. Lee, and S. M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, vol. 2018-July. Institute of Electrical and Electronics Engineers Inc., 7 2018, pp. 1492–1499.

[19] J. Napieralla, "Considering webassembly containers for edge computing on hardware-constrained iot devices," Master's thesis, Blekinge Institute of Technology, 2020. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:bth-20112

[20] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," April 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3302505.3310084

[21] Apache, "Apache openwhisk," accessed: 2021-03-17. [Online]. Available: https://openwhisk.apache.org/

[22] Wasmer, Inc, “Wasmer - the universal webassembly runtime,” accessed: 2021-03-30. [Online]. Available: https://wasmer.io/

[23] Harbor, “Harbor,” accessed: 2021-03-30. [Online]. Available: https://goharbor.io/

[24] “Raspberry pi 4 hardware specs and comparison - pi supply maker zone,” accessed: 2021-04-19. [Online]. Available: https://learn.pi-supply.com/make/raspberry-pi-4-hardware-specs-comparison/

[25] Shelly Cloud. Allterco Robotics LTD , “Shelly plug s - shelly cloud,” accessed: 2021-04-19. [Online]. Available: https://shelly.cloud/products/shelly-plug-s-smart-home-automation-device/