# David vs. Goliath: Comparing Performance of fork() in Raspberry Pis and Servers

*Mohammad Ayaan Hashim*

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

# Abstract

This dissertation investigates the comparative performance of the fork() system call across two distinct computing platforms: the Raspberry Pi (ARM64 architecture) and traditional server-grade hardware (x86 architecture). Focused on uncovering architectural efficiencies and bottlenecks, this study thoroughly examines various performance metrics, including CPU clock times, cache behaviour, and instruction throughput, under both single and concurrent execution scenarios. Through a series of methodically designed experiments, this research highlights the Raspberry Pi's surprising efficiency in managing single-threaded tasks, despite its lower computational power relative to the server. Concurrent execution analysis reveals significant insights into each platform's handling of multitasking, with the Raspberry Pi facing challenges that underscore its limitations in a multitasking context. By delving into the fork() syscall's behaviour, this work identifies critical hotspots and architectural differences that influence performance. The findings not only contribute to a deeper understanding of ARM64 versus x86 architectural performance nuances but also propose a foundation for future optimisation strategies aimed at enhancing system design and application deployment across diverse hardware platforms. This research advances the Democratization of computing infrastructures, advocating for innovative, efficient, and adaptable solutions in the face of evolving serverless and edge computing paradigms.

## Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

## Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Mohammad Ayaan Hashim*)

# Acknowledgements

I would like to express my deepest gratitude to my supervisor, Yuvraj Patel, whose expertise, insightful feedback, and constant encouragement were invaluable to the completion of this dissertation. His unwavering support and push for excellence greatly enriched my research journey and motivated me to strive for quality in every aspect of my work.
I would like to thank my family for their unwavering love and support throughout this research. My Mummy, Papa and Bhai, whose belief in me has been a constant source of motivation and strength.

# Table of Contents

# Chapter 1

# Introduction

In this digital renaissance age, the relentless pursuit and advancement in data generation and processing demands has spotlighted the limitations of conventional computing infrastructures. The emergence of serverless computing paradigms has introduced a revolutionary shift towards optimising resource allocation, operational costs, and scalability. However, this innovation isn't without its challenges, particularly in latency-sensitive applications. This scenario underscores the need for an avant-garde computing solution that not only align with the principles of serverless architecture but also mitigate its inherent drawbacks.

This dissertation delves into a comprehensive exploration of leveraging Raspberry Pi clusters as an innovative substitute for traditional server setups, thereby contributing to the evolving narrative of computing infrastructures. This study also aims to dissect the viability, performance, and efficiency of Raspberry Pi clusters within the field of serverless and edge computing. Stemming from a critical need to **"democratize"** computing resources, ensuring they provide accessibility, efficiency, and sustainability for everyone.

The research is anchored on a methodical evaluation of the Raspberry Pi 4B model's performance, focusing on its capability to execute the fork system call—a key operation in serverless computing frameworks. Through a carefully designed experimental setup, this dissertation contrasts the Raspberry Pi's performance against traditional server-grade hardware, delving into performance metrics, hotspots identification, and the execution efficiency of the fork system call. Through this structured approach, the dissertation endeavours to contribute meaningful insights to the discourse on computing infrastructures, advocating for innovative solutions that challenge the status quo and pave the way for future advancements.

The following shows a roadmap of this dissertation:

- **Chapter 2: Background and Literature Review** dives into the foundational concepts underlying the study. Here, we explore the evolution of computing infrastructures, with a particular focus on serverless and edge computing paradigms. The chapter reviews the current state of research, highlighting the significance of the fork() system call in serverless architectures and presents the Raspberry Pi as a promising contender in edge computing. Through a critical analysis of existing literature, this chapter establishes the context for our research and justifies the need for a comparative performance evaluation.

- **Chapter 3: Motivation and Objectives** delves deeper into the rationale behind focusing on the Raspberry Pi for edge computing solutions and the specific role of the fork() system call in evaluating its viability against traditional servers. This chapter elucidates the research goals and outlines the objectives designed to guide the experimental investigations, providing clarity on the scope and ambition of the study.
- **Chapter 4: Methodology** provides a comprehensive overview of the research approach and the analytical framework adopted in this study. It explains the rationale behind the selection of the Raspberry Pi and traditional server-grade hardware for performance comparison, outlines the criteria for evaluating the fork() system call, and details the methodological principles guiding the experimental design. This chapter establishes the theoretical and practical foundation upon which the experimental investigation is built, including the selection of specific performance metrics and the justification for the chosen research methods.
- **Chapter 5: Experiments** presents the experiment types and setups integral to the study. This chapter discusses the parameters and configurations used during the experiments, both in single and parallel execution contexts. It details the allocation of memory for the tests and the rationale behind using certain subtypes of experiments for the Performance Measurement Analyses and the Hotspot Investigation.It also lays the groundwork for the rigorous analysis of performance in relation to the fork() system call, setting the scene for the nuanced examination of results
- **Chapter 6: Results and Discussions** synthesises the experimental outcomes, as well as provides a critical interpretation of the findings within the broader context of computing infrastructure optimisation. This chapter not only compares the performance of Raspberry Pi and traditional servers but also discusses the implications of these results for serverless and edge computing practices.
- **Chapter 7: Conclusions and Future Works** draws together the key conclusions derived from the research, highlighting the contributions of this study to the field of computing infrastructures. It reflects on the implications of the findings for future technology development and outlines potential directions for further research, aiming to inspire continued exploration into efficient, adaptable computing solutions.

# Chapter 2

# Background and Literature Review

## 2.1 Background

### 2.1.1 Raspberry Pi

#### 2.1.1.1 Overview

The Raspberry Pi is a series of low-cost, credit card-sized single-board computers developed by the Raspberry Pi Foundation [Severance, 2013]. Primarily introduced with the goal of promoting teaching basic computer science in schools and in developing countries, has seen adoption spread far beyond the initial educational purposes. Equipped with a range of processors, from single-core ARMv6 to quad-core ARMv8, and various RAM configurations, Raspberry Pis are capable of undertaking many tasks traditionally reserved for desktop computers.

#### 2.1.1.2 Role

Raspberry Pi's role in computing has evolved significantly. Its compact size, affordability, and reasonable computational power makes it an ideal tool for hobbyists, educators, and professionals for tasks ranging from learning programming to hosting websites and servers. In the context of edge computing, Raspberry Pi serves as a powerful and cost-effective edge device that can process data locally, reducing latency, and minimising the need for constant cloud connectivity.

### 2.1.2 Servers

#### 2.1.2.1 Overview

Servers are not defined by specific hardware but by the role a computer serves within a network, providing resources, data, services, or programs to other computers, known as clients. This essential function facilitates multiple users or systems to access shared resources efficiently, underpinning the client-server architecture. Servers in their traditional form are powerful rack-mounted units designed for data centers adapting to diverse operational needs and application demands. Though keeping this in mind we refer to a “Server” as one in its traditional setup.

#### 2.1.2.2 Role and Evolution

Conventionally, servers have been the backbone of corporate networks and the internet, hosting websites, applications, and databases. With the advent of cloud computing, the role of servers has extended, enabling scalable, on-demand computing resources over the internet. This evolution has paved the way for serverless computing, where the cloud provider dynamically manages server allocation, allowing developers to focus on building and running applications without managing the underlying hardware.

### 2.1.3 Serverless Computing

#### 2.1.3.1 Definition and Evolution

Serverless computing, often associated with the Function-as-a-Service (FaaS) paradigm, refers to a cloud-computing execution model where the cloud provider dynamically manages the allocation of machine resources. Contrary to its name, serverless computing does involve servers, but the responsibility of managing these servers and resources is shifted from the consumer to the provider. This model allows developers to focus solely on the individual functions of their application code.

#### 2.1.3.2 Role in Modern Applications

Serverless computing has transformed the way applications are developed, deployed, and scaled. Enabling developers to build applications that automatically scale with demand, without the need to manage underlying infrastructure. In edge computing environments, serverless computing can optimise resource utilisation by executing functions closer to the data source (also referred to as Edge), thereby improving response times for latency-sensitive applications.

### 2.1.4 Edge Computing

#### 2.1.4.1 Concept and Importance

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed (or utilised), to improve response times and save bandwidth. The essence of edge computing lies in minimising the distance between the data source and the processing power, thereby reducing latency and enhancing the efficiency of data processing.

#### 2.1.4.2 Role

In an increasingly connected world, where devices generate vast amounts of data, edge computing plays a critical role in enabling real-time analytics and decision-making. By processing data locally, at the edge of the network, rather than relying solely on centralized servers, edge computing supports a wide array of applications, from Internet of Things (IoT) devices to autonomous vehicles and smart cities. This paradigm shift not only speeds up data processing but also addresses privacy and security concerns by localizing data.

## 2.2 Literature Review

### 2.2.1 Evolution of Edge Computing with Raspberry Pi

The advent of Raspberry Pi has marked a significant milestone in the evolution of edge computing. With its low cost, energy efficiency, and considerable computational power, Raspberry Pi presents a compelling case for widespread deployment in edge computing networks. The use of Raspberry Pi in applications such as offline media servers [Jadhav and Malode, 2019] demonstrates its ability to handle data-intensive tasks with limited resources, underscoring the potential for Raspberry Pi-based systems to support a wide range of edge computing applications. However, performance assessments reveal variability based on workload types, network conditions, and hardware configurations, pointing to the necessity for careful system design and optimisation to harness the full potential of Raspberry Pi in edge environments.

### 2.2.2 Serverless Computing at the Edge: Opportunities and Challenges

Serverless computing models, particularly Function-as-a-Service (FaaS), promise to simplify application deployment and scaling in edge computing by abstracting away infrastructure management complexities. However, the integration of serverless models with edge computing introduces new challenges, primarily due to the heterogeneous and resource-constrained nature of edge devices. Benchmarking efforts, such as BenchFaaS [Carpio et al., 2023], have highlighted the impact of these challenges on serverless function performance, including increased overheads and variable execution times. These findings emphasise the need for specialised frameworks and deployment strategies to adapt serverless computing models to the edge environment effectively.

### 2.2.3 Raspberry Pi and Edge-Cloud Collaboration

The Raspberry Pi Edge-Cloud Collaboration Framework (RPECCF) [Zhang et al., 2020] represents a novel approach to achieving seamless integration between edge devices and cloud resources. By leveraging Raspberry Pis as edge nodes, the RPECCF aims to balance computational loads between the edge and the cloud, optimising resource allocation and reducing latency for time-sensitive applications. Experimental evaluations of this framework demonstrate the Raspberry Pi's capabilities in handling edge computing tasks while interacting efficiently with cloud services. This collaboration between edge and cloud resources highlights the potential for Raspberry Pis to serve as vital components in distributed computing architectures, facilitating improved application performance and resource utilisation.

### 2.2.4 Towards Optimised Edge Computing with Raspberry Pi

The collective insights from research on Raspberry Pi's performance in edge computing and the optimisation of serverless models reveal a path toward more efficient edge computing infrastructures. By addressing the challenges identified through benchmarking and performance analysis, such as fork operation optimisation and serverless function deployment, it is possible to enhance the throughput and scalability of edge computing networks. Continued innovation and experimentation with Raspberry Pi and serverless models will be crucial in unlocking new capabilities and achieving the full promise of edge computing.

### 2.2.5 Fork implementation Caused Bottlenecks

Through the thorough discussions in the paper 'A fork() in the Road' [Baumann et al., 2019], the inefficiency of the current implementation of the fork syscall is heavily discussed, touting it as an **"anachronism"**. The paper also highlights issues related to the use of fork, noting its lack of thread safety, lack of composition, inefficiency, un-scalability, and the introduction of security concerns. Additionally, it points out how fork limits the innovation capacity of OS researchers and developers, as any new abstraction must be meticulously special-cased for. Since systems that provide provide for the fork call, if not a system based on monolithic kernel, are forced to lazily duplicate per-process state.

These problems are especially relevant and important given the findings from [Atlidakis et al., 2016], that reports the fact that **1304** of Ubuntu packages, about 7.2% of the total number, use the fork() call. This is along with the fact that almost all of the UNIX-shells, database servers (Oracle, PostgreSQL, Apache, etc.), Redis (a key-value store), Node.js and even Google Chrome, utilise fork in some capacity.

The gravity of these findings have lead recent developments in a better approach not only for the fork syscall, but also to improve the current Copy-on-Write mechanisms. Developments like On-Demand Fork [Zhao et al., 2021], introduces a novel approach to copying page tables between the parent and child at fork time. This is done specifically in small chunks, "on-demand" when handling page faults. Such a novel approach showed a marked improvement in the fork execution, especially in applications with a larger memory footprint.

# Chapter 3

# Motivation And Objectives

## 3.1 Motivation

The exponential growth of data and computational needs in various sectors, from academic research to industry applications, has significantly underscored the necessity for efficient, cost-effective, and sustainable computing infrastructures. Traditional server setups, while robust and powerful, often present challenges in terms of scalability, energy consumption, and capital investment, particularly for small to medium-sized enterprises (SMEs) and educational institutions. To combat this, many developments have been made, especially in reagrds to serverless architecture. This emergence of serverless architecture represents a paradigm shift towards minimising the operational complexities and costs associated with physical servers. Serverless computing abstracts server management and dynamically manages the allocation of machine resources, it promises scalability, efficiency, and a pay-as-you-go pricing model.

However, the introduction of the Serverless Architecture does not solve all the challenges. Despite its benefits, serverless computing, by its very design, can introduce latency issues for data-intensive applications due to its reliance on centralised cloud data centers/servers. This latency becomes a bottleneck for real-time applications, underscoring the emerging necessity for computing paradigms that can process data closer to where it is generated or needed. The shift towards edge computing represents a strategic move to address these challenges, promising reduced latency, enhanced data processing speeds, and improved overall application performance by decentralising computation and bringing it closer to the edge of the network, where users interact with the systems. Due to the progression towards the convergence of two major industry trends as outlined in Peterson et al. [2019]. The first being the expansion of cloud services out of datacenters and towards the network edge and the second being network operators transitioning their access networks from proprietary hardware to virtualised software running on commodity servers, switches and access devices. Thereby the paper explores the need for "Democratizing the Network Edge", the essence of which means to ensure the aversion of monopoly of large cloud providers over edge computing infrastructure. By making edge computing resources more accessible to a wider array of developers and organisations, innovation at the edge is spurred, creating a more vibrant and competitive ecosystem. Such an open edge ecosystem would not only foster innovation but also ensure that the benefits of edge computing—lower latency, improved bandwidth usage, and enhanced privacy and security—are realised across various sectors.

This brings us to the Raspberry Pi, a puny Single-Board Computer (SBC), which surprisingly comes up as a promising aspect in realising the democratised network edge, offering a compact, energy-efficient, open, and affordable edge. However, the feasibility of deploying Raspberry Pi clusters as server replacements relies on their ability to perform well in serverless applications. The reliance of serverless applications on exploiting the Copy-on-Write (CoW) principle that the system call fork provides plays a pivotal role in performance. For example, applications like Redis [red], a popular in-memory key value storage, designed in a way to provide high throughout and minimise latency. This in-memory persistence is handled through using fork. Redis handles incoming requests by allowing the in-memory index and data structures only and periodically invokes fork, this basically allows the entire in-memory content to be duplicated to the child process. The original inbound request during this time would continue processing. While the child process uses its memory as a snapshot, serialising the in-memory datastructures as files, this ensures the persistence of the in-memory snapshot. So that upon a system crash, Redis can be restored through a previous snapshot.

The motivation for this thesis stems from the need to explore alternative computing infrastructures that not only align with the evolving landscape of serverless and edge computing but also challenge the current dominance of conventional server setups and centralised cloud data centers. This exploration is critical in addressing the inherent limitations these traditional systems impose, particularly regarding latency, scalability, and openness. The Raspberry Pi, with its compact, energy-efficient, and cost-effective design, presents itself as an extremely interesting candidate, especially when deployed in clusters for potential use as edge servers or microservers in a serverless architecture.

Another factor influencing the choice of Raspberry Pi is the ability to leverage a large number of CPU cores at a relatively low cost. While traditional servers offer higher CPU clock speeds, they are limited in the number of CPU cores, typically maxing out at 64 cores for single-socket boards, 128 cores for dual-socket, and 256 cores for quad-socket configurations. In contrast, a cluster of Raspberry Pis, with each having a quad core, can provide access to over 400 CPU cores at a comparable price point, making it an attractive alternative for highly parallelisable workloads that can effectively utilise a large number of lower-frequency cores.

## 3.2 Objective

The overarching goal of this thesis is to conduct a comprehensive evaluation of the Raspberry Pi, particularly the 4B model. Specifically, this thesis aims to:

- **Investigate the Raspberry Pi's Performance with a Focus on the Fork System Call**: This objective involves a detailed examination of the Raspberry Pi 4B's ability to execute the fork system call efficiently. Given the fork call's critical role in serverless computing architectures—especially in processes requiring rapid duplication for concurrent task handling—this analysis is pivotal. The investigation will encompass various performance metrics, including but not limited to, execution time, memory usage, and process handling capacity, to provide a holistic view of the Raspberry Pi's capabilities in this regard.

- **Compare the Fork Performance on Raspberry Pi Against Traditional Server-Grade Hardware**: To contextualise the Raspberry Pi's performance, a comparative analysis with traditional server-grade hardware will be conducted. This comparison aims to highlight the Raspberry Pi's efficacy and viability as a microserver alternative within edge computing environments.
- **Identify and Analyse Hotspots in the Execution of the Fork System Call on Raspberry Pi**: This objective delves into identifying performance bottlenecks or "hotspots" during the fork system call execution on the Raspberry Pi. Through profiling and tracing analysis, this research seeks to uncover any computational or resource allocation challenges that may impact the Raspberry Pi's performance. Understanding these hotspots is crucial for optimising the Raspberry Pi's system architecture and for proposing enhancements that could mitigate such bottlenecks.

# Chapter 4

# Methodology

## 4.1 Introduction to Methodological Approach

To explore the capabilities of Raspberry Pi clusters as a feasible alternative to conventional server infrastructures, a rigorous and structured methodological approach is essential. This chapter explains the comprehensive strategies and procedures implemented to evaluate the performance of the Raspberry Pi, with a special focus on the efficiency and effectiveness of the fork syscall. The methodologies employed are designed to not only quantify the performance metrics of the Raspberry Pi in server-like operations but also to offer a comparative insight against traditional server setups. This comparative analysis is crucial, as it situates the Raspberry Pi within the broader context of current computing solutions, providing a clear perspective on its potential as a server replacement.

This pursuit of evaluating the Raspberry Pi's capabilities necessitates not just a broad methodological framework but also a deep dive into the specifics of each technique and tool employed. Each methodological choice, from the selection of hardware configurations to the detailed performance metrics, is driven by a commitment to rigour, relevance, and the pursuit for actionable insights.

Aiming to go beyond the conventional performance benchmarks to delve into how the Raspberry Pi, with its ARM-based architecture, stands up against the computational prowess of traditional x86 servers. A thorough probe into the heart of the Copy-on-write principle, brought about due to the focus on the fork syscall. This principle plays a critical role in both server operations and serverless applications.

The novelty of this methodological approach lies in its comprehensive framework, combining customised workloads, precise performance metrics and an in-depth investigation into the performance hotspots.

## 4.2 Experimental Setup

To validate the findings and establish a robust basis for comparison, the experimental setup was meticulously designed to focus on the performance evaluation of a singular Raspberry Pi 4B against a traditional server-grade hardware setup, provisioned through CloudLab [Duplyakin et al., 2019]. It was also ensured that the underlying workload simulating the fork syscall was kept consistent between the two. This targeted approach allows for an in-depth analysis of the fork syscall's efficiency and effectiveness within the Raspberry Pi versus the traditional server.

### 4.2.1 Hardware Employed

#### 4.2.1.1 Raspberry Pi 4B

1. **CPU:** 64-bit Quad-Core Cortex-A72 processor @ 1.5 GHz
2. **CPU architecture:** ARM 64-bit
3. **RAM:** 8GB LPDDR4 @ 3200 Mhz
4. **Cache:**
   - L1d: 128 KiB
   - L1i: 192 KiB
   - L2: 1 MiB
5. **Kernel:** 6.5.0-1012-raspi

#### 4.2.1.2 Server

1. **CPU:** Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
2. **CPU architecture:** x86_64
3. **RAM:** 251 GiB DDR3 @ 1600 MHz (1333 MT/s)
4. **Cache:**
   - L1d: 640 KiB
   - L1i: 640 KiB
   - L2: 5 MiB
   - L3: 50 MiB
5. **Kernel:** 5.15.0-86-generic

### 4.2.2 Workload and Environment Configurations

To accurately assess the fork syscall performance, a custom workload in C was developed, paying attention to the environment setting. These workload configurations are meticulously designed to evaluate how the systems manage process creation and memory utilisation under conditions that simulate everyday server-like operations. The key components of the workload include:

1. **CPU Affinity Settings:** At the beginning, the environment is configured through explicitly setting the CPU affinity using `CPU_ZERO(&mask); CPU_SET(cpu_to_set1,& mask);` to bind the executing process to a specific CPU core ( core cpu_to_set1). This initial setting aims to simulate a controlled environment for the initial operations of the workload.

2. **Memory Allocation:** This configuration allocates a specified amount of memory (in MB) passed in as an argument:

   ```
   int mem_size = atoi(argv[1]);
   size_t size = (1024 * 1024 * mem_size);
   void *buffer = mmap(NULL, size, PROT_READ | PROT_WRITE,
       MAP_PRIVATE | MAP_ANONYMOUS, -1, -0);
   ```

   Utilising 'mmap' to allocate this memory, ensuring it is accessible and writable, thereby simulating the memory footprint of active server processes.

3. **Memory Initialisation:** Following allocation, memset is used to initialise the allocated memory space (`memset(buffer, 0, size);`), representing the preparatory stage of process workload.

4. **Clearing Cache:** Before executing the workload, the the page cache, dentries, and inodes of the environment is cleared with `system("sudo␣echo␣3␣>␣/proc/sys/vm/drop_caches");` to minimise the impact of cached data on the workload's performance measurement.

5. **Optional CPU Affinity Adjustment:** Just before executing the workload, the CPU affinity is adjusted to a different core using `CPU_ZERO(&mask); CPU_SET(cpu_to_set2, &mask);`. This step is critical for examining the effects of CPU migration, aiming to bypass the influence of CPU cache on the simulation.

6. **[Workload] Fork Execution:** The workload consists of the fork syscall which is executed with `fork_pid = fork();` on the previously configured environment, creating a child process to assess the fork operation's performance.

7. **Process Exit and Cleanup:** Child processes created by the fork exit immediately to prevent interference (`if (fork_pid == 0)_exit(0);`), while the parent process waits for the child's termination (`waitpid(-1, NULL, 0);`). Following this, the allocated memory is released (`munmap(buffer, size);`), ensuring a clean state for subsequent iterations.

### 4.2.3 Experiment Types

The described workload and environment configurations enabled two types of experiments, one that kept the CPU affinity the same throughout the execution (No Migration type), and another that would switch the CPU affinity in the 'Optional CPU Affinity Adjustment step' (Migration type). This allows investigation into the potential impact of CPU migration on the workload's (fork) performance. Permitting a focused analysis of performance, excluding the effects of CPU cache, thereby providing insight into the raw computational capabilities of the Raspberry Pi when compared to traditional server hardware.

These experiments are differentiated by their execution context — parallel/concurrent versus singular execution. This approach enables a nuanced analysis of each system's ability to handle multitasking and process management under varied operational loads.

#### 4.2.3.1 Parallel Execution

**4.2.3.1.1 With CPU Migration** This subtype involves running the workload on cores 0 and 2 initially, and after memory allocation, switching the CPU affinity to 1 and 3 respectively. Doing so simulates a scenario where a process might be migrated across cores in a multicore execution.

**Purpose:** To assess the impact of CPU migration on the performance of concurrent operations, particularly looking at potential variations in execution times or resource utilisation due to the CPU cache effects or scheduling overhead.

**4.2.3.1.2 Without CPU Migration** In this, the workload is executed on the four cores for the Pi, and the first four cores for the server, without changing the CPU affinity, ensuring that each process remains on its initially assigned core throughout the experiment.

**Purpose:** To evaluate the baseline performance of parallel executions when each process is consistently executed on the same CPU core, providing a control scenario for comparison with the migration subtype.

#### 4.2.3.2 Singular Execution

**4.2.3.2.1 With CPU Migration** In this, the workload is initially assigned to a single CPU core, with the affinity changed to another core after memory allocation, mimicking a single-task migration scenario.

**Purpose:** To explore how migrating a single process between cores affects the execution efficiency of the fork syscall, highlighting the effects of such migrations in a controlled, single-core context.

**4.2.3.2.2 Without CPU Migration** In this, the workload runs on a single core with a fixed CPU affinity, ensuring no migration occurs, to isolate the performance of the fork syscall in a static core assignment.

**Purpose:** To establish a performance benchmark for the fork syscall on a single core without the influence of CPU migration, serving as a point of comparison to understand the implications of migration in similar scenarios.

## 4.3 Performance Measurement Methodology

### 4.3.1 Overview of Performance Investigation

For a thorough investigation into the performance characteristics of both the Raspberry Pi and the server, particularly focusing on the fork execution of the workload, the 'perf_event_open' API [Linux Man-pages Project, 2024] was utilised to ensure inline measurements. This choice was driven by the API's versatility in offering detailed insights into a wide range of hardware and software events, facilitating a nuanced performance analysis in both types of the experiment.

### 4.3.2 Event Selection for Measurement

To ensure a comprehensive analysis, we selected a broad set of both hardware-specific and architecture-agnostic software events. This dual approach is not merely about accumulating data but is strategically aimed at testing specific hypotheses that would help understand the performance characteristics of the Raspberry Pi and the server in executing the fork syscall.

A holistic analysis of fork's performance is enabled through integrating both hardware and software sets of metrics. By juxtaposing hardware-specific insights with software-level observations, a comprehensive performance narrative that accounts for the interaction between system architecture, memory management, and process scheduling can be constructed.

Events measured encompassed all the common pre-included ones in the perf_event_open. This selection made available hardware events such as "CPU_CYCLES", "INSTRUCTIONS", "CACHE_REFERENCES", along with software events like "CPU_CLOCK", "PAGE_FAULTS", etc. Additionally, the Cache events such as "CACHE_L1D_ACCESS", "CACHE_L1D_MISS", etc. were also included.

### 4.3.3 Performance Measurement Implementation

The implementation of these measurements involved scripting the perf_event_open API calls to capture the specified events during the fork syscall execution. Special attention was paid to the experimental conditions to ensure consistency across runs, particularly in scenarios exploring the effects of CPU migration. The CPU affinity settings, both before and after memory allocation, were carefully managed to assess the impact of process migration across CPU cores.

## 4.4 Identifying Performance Hotspots

### 4.4.1 Overview

To gain in-depth insights into the execution flow and performance bottlenecks of the fork() syscall, 'ftrace' [Bird, 2009] was utilised for its robust tracing capabilities. Inline tracing was strategically implemented by inserting trace markers to denote the start and end points of the fork execution step of the workload. This approach allowed for the isolation and precise timing of syscall execution, providing a clear demarcation for further investigation.

### 4.4.2 Implementing Trace Markers

Trace markers were integrated directly into the C code to flag the critical sections surrounding the fork syscall. By employing `'sprintf'` to format strings indicating "Before fork" and "After fork" events along with memory size and CPU settings, and then writing these markers to a trace file, a detailed execution timeline was established.

### 4.4.3 Shell Script Automation for Tracing Setup

To streamline the tracing process and ensure comprehensive data capture, shell scripts were developed to configure the ftrace settings and initiate the tracing sessions. This automation facilitated the efficient collection of trace data across different CPU cores and experimental conditions.

1. **Tracing Configuration:** The scripts set ftrace to use the 'function_graph' tracer, enabling detailed function call graphs. Furthermore, the tracing granularity was further refined by enabling several trace options: 'funcgraph-tail', which appends the function name when the exit/return event occurs, and funcgraph-abstime, set to capture the absolute timestamp of each trace event, thereby offering a precise temporal context to the tracing data.
2. **Configuring Buffer Size:** To accommodate the extensive volume of data generated during the tracing, the buffer size was configured to 51200KB (per CPU tracer). This adjustment was crucial for ensuring that no significant trace events were omitted due to buffer overflows, thereby preserving the integrity of the data collected for analysis.
3. **Targeted Function Graphing for Fork Syscall Analysis:** By focusing the tracing on the entry point for the fork syscall, **`'__arm64_sys_clone'`** in the Raspberry Pi and **`'__x64_sys_clone'`** in the server, the scripts provided a concentrated view of only the syscall's performance.

## 4.5 Limitations of the Raspberry Pi

While the Raspberry Pi is a robust platform for a variety of applications and demonstrates it capability in the hobbyist community. It presents specific challenges in the context of performance measurement for server-like operations, particularly when utilising tools such as perf and other performance monitoring utilities. These limitations are critical to understanding the scope and for comparative analysis conducted within this study.

### 4.5.1 Limited Support for Perf Counters and Events

#### 4.5.1.1 Kernel and Architecture Specificity:

The 'perf' tool's dependence on the kernel and architecture leads to significant limitations for the Raspberry Pi. Many of default events and counters typically available within **`'perf list'`** on x86 architectures lack support on the Raspberry Pi's ARM architecture. This absence greatly restricts the ability to monitor a comprehensive set of performance metrics directly comparable to those of traditional servers.

Fundamental counters for **`'perf stat'`** such as `'LLC-loads'`, `'LLC-load-misses'`, `'dTLB-loads'`, `'iTLB-loads'`, `'L1-dcache-prefetches'`, and `'L1-dcache-prefetch-misses'` are not supported on the Raspberry Pi. The lack of these metrics posed a challenge in extensively analysing cache efficiency and memory access patterns, which would be crucial in evaluating the impact of cache performance on fork.

#### 4.5.1.2 Limited Command Support:

Many of the basic popular and useful perf commands like **`'perf mem'`**, **`'perf trace'`**, **`'perf probe'`**, and **`'perf timechart'`**, even though theoretically possible, do not in practice support the Raspberry Pi. This greatly hinders the ability to perform in-depth memory, tracing, and timing analysis, along with further removing the ability to make new tracepoints. All of which would have greatly saved time in performing detailed performance analysis. As a result, necessitated the use of more granular, lower-level implementation of 'perf' through the **`'perf_events_open'`** API.

#### 4.5.1.3 Issues with Compatibility in Default perf_event_open API Events:

Speaking of the primary reason for diverting to the use of the 'perf_event_open' API, quite a lot of the default events that are provided by the granular implementation were also missing. These included the following hardware counting events ('COUNTS_HW') that were not supported

1. **"PERF_COUNT_HW_BRANCH_INSTRUCTIONS"**
2. **"PERF_COUNT_HW_REF_CPU_CYCLES"**
3. **"PERF_COUNT_HW_CACHE_LL"**
4. **"PERF_COUNT_HW_CACHE_BPU"**
5. **"PERF_COUNT_HW_CACHE_NODE"**

Furthermore, the predefined cache counting events ('COUNT_HW_CACHE') did not support counting the prefetch access operations (**"CACHE_OP_PREFETCH"**). While the "**CACHE_DTLB**" and the "**CACHE_ITLB**" did not support the counting the cache result accesses either ("**CACHE_RESULT_ACCESS**").

While these did not completely prohibit the ability to include these counters, additional research and implementation was required by utilising the Raw PMU event codes from the CPU Technical Reference Manual [ARM, 2016].

### 4.5.2 Performance Measuring Tool incompatibility

#### 4.5.2.1 SystemTap Incompatibility

Despite efforts to compile the kernel with the required configurations, SystemTap was non-functional on the Raspberry Pi. This incompatibility greatly hampered the approach to kernel-level tracing and monitoring. Thereby necessitating a pivot to use 'Ftrace' instead along with further custom scripting in order to achieve a SystemTap equivalent.

#### 4.5.2.2 Required Kernel Recompilation

Enabling most performance monitoring tools on the Raspberry Pi requires extensive debugging configurations and kernel recompilation—a process notably more cumbersome on the ARM architectures than on x86. This complexity adds a significant barrier to utilising a wide range of performance analysis tools.

#### 4.5.2.3 General Community Support

The general community support for Performance Monitoring Units (PMUs) and performance measurement tools on the ARM architecture is not as extensive as that for x86. This discrepancy affects the availability and support for ARM-compatible performance measuring and analysis tools. This therefore limited the breadth of up-to-date community-developed tools that could be employed in the study.

### 4.5.3 Conclusion to Limitations

This need to constantly adapt and pivot from established methodologies to a more bespoke, lower-level implementation highlighted the dynamic nature of decision-making throughout this study. The exploration of various tools and methodologies, some of which might have not have been considered in an x86-centric study, became a pivotal aspect in the research process. It underscored the importance of flexibility and creativity in navigating the limitations posed by the Raspberry Pi's architecture.

These methodological pivots, while challenging, enriched the study by broadening the technical repertoire and deepening understanding of ARM-based system performance. They underscore the broader need for support for performance analysis tools to the unique requirements of ARM architecture.

# Chapter 5

# Experiments

Following the description of the experiment types from 4.2.3, the major types of the experiment, the parallel and single executions stays constant between the two analyses.

While the entire suite of the subtypes of the experiments are used in the Performance Measurement Analyses, the Hotspot Investigation only employes the migration subtype in a single execution manner.

The memory allocations done for both of the Analyses were: 1, 8, 16, 32, 64, 128, 256, 512 and 1024 MB, each of these were the memory configuration of the environment 4.2.2.

## 5.1 Performance Specific Analyses

### 5.1.1 Experiment Setup

Each memory allocation for every experiment was executed 1000 times. Conducting 1000 runs for each memory allocation scenario not only ensures the statistical significance of the performance analysis but also highlights the variance in performance across these runs. This comprehensive approach illuminates how performance metrics fluctuate under identical conditions, offering insights into the stability and reliability of different memory allocation strategies.

### 5.1.2 Visualisation

To effectively visualise the results from each experiment, 'Matplotlib' library was utilised. The results were categorised as either **'Migration'** or **'No Migration'**. The data visualisation included the average of 1000 runs along with the maximum and minimum observations for each event. The plots provided a comparative analysis, showcasing both the baseline—representing single execution—and the concurrent execution outcomes for the Raspberry Pi and the server.

## 5.2 Hotspot Investigation Analyses

### 5.2.1 Data Extraction and Analysis

Following the collection of trace data for each of the aforementioned memory allocations, a custom analysis python script was implemented to decode the results and extract key performance metrics into a CSV format. This structured data included the function name, duration, parent function, depth from the sys_clone call, and the timestamp of execution completion.

1. **Decoding Trace Results:** After tracing, the raw output captured by ftrace underwent a meticulous decoding process. This step involved parsing the trace files to isolate and analyse each function call that occurs within the sys_clone function call. Key metrics such as the duration of each function and its hierarchical depth relative to the sys_clone were carefully extracted. This detailed parsing was instrumental in pinpointing areas within the sys_clone execution that could potentially act as performance bottlenecks, thereby revealing the critical performance hotspots.

2. **CSV Data Extraction:** The next phase involved transforming the decoded trace information into a structured CSV format, focusing on specific data points: Function Name, Duration of the Function, Parent Function, Depth of the Function from the initial sys_clone call, and the Timestamp of the function's execution.

### 5.2.2 Visualisations

#### 5.2.2.1 FlameGraph

Due to the nature of the overwhelming volume of data returned by FTrace, even after the extraction performed. The visualisations made needing to successfully depict the overall performance for each of the functions called, a FlameGraph, inspired by BrendanGregg [Gregg], needed to be made. Due to the lack of support of making these for Ftrace through the official tools. This was done using **'plotly'** to make it interactive.

#### 5.2.2.2 Stack Graph

To complement the Flamegraph's high-level insights into execution hotspots, an additional, more detailed helper graphs were created. These were done by plotting the duration tied to memory use within each function at a given depth. Special attention was directed towards omitting functions with minimal deviation across various memory allocations, ensuring a focus on those functions that exhibited variation over memory allocations. Thereby only plotting functions execution that were dependant on the memory allocations. Additionally, the duration for the function was kept in $log_{10}$.

# Chapter 6

# Results and Discussions

## 6.1 Performance Results

Here, for all the concurrent plots, we only depict one of the processes' performance to show the affect that all of the

### 6.1.1 No Migration Type

In this section we assess the results of the Performance Measurement Experiments, specifically in the sub-type in which the CPU affinity remained the same throughout.

#### 6.1.1.1 CPU Clock

Starting our investigation for the no migration case, we first look at the CPU clock trends in order to observe basic trends when it comes to the execution times. This can be found in Fig.6.1.

- The Pi's Baseline exhibits the lowest software CPU clock times, suggesting that when operating without concurrent processes, the Pi's architecture is highly efficient.
- When comparing concurrent execution times, the Pi's concurrent performance shows it's marginally slower than the others. This small difference could indicate that while the Pi is not optimised for concurrency, it doesn't significantly lag behind when dealing with parallel processes. This is especially apparent in the higher memory allocations.
- The Server's Baseline and concurrent performances are close, indicating its architecture is not heavily influenced by concurrency. This could be due to better core-to-core communication or a more advanced scheduling system that manages parallel processes more effectively.
- Interestingly, the Server does not exhibit a performance dip or significant advantage in either mode, suggesting that it is engineered to handle a mix of tasks (both sequential and parallel) with consistent efficiency.

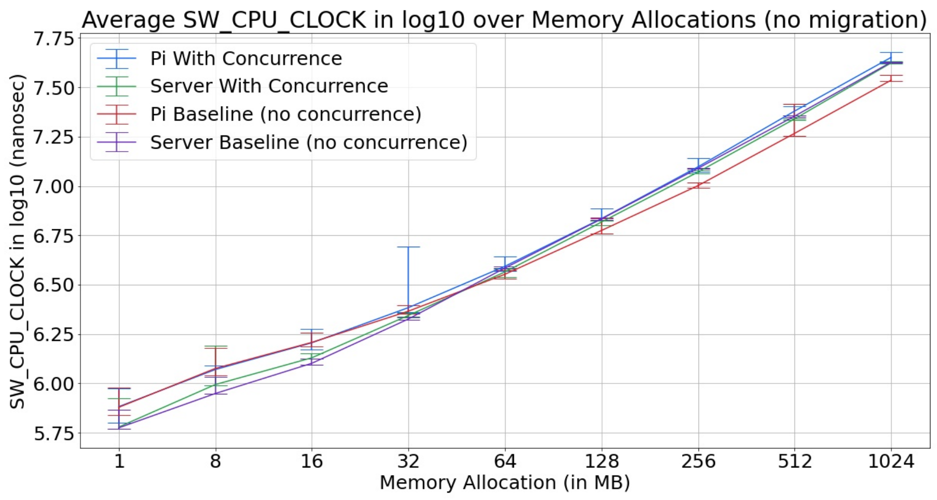


Figure 6.1: Figure showing the Average CPU Clock per Memory allocation along with the min and max variation

- The similar performance trends of the Server in both Baseline and concurrent variants could also imply that it's built to scale with the demands of multitasking environments.
- The convergence of performance lines around the 64MB memory allocation could suggest that each system's cache management strategy begins to have a less differentiated impact on performance. This convergence hints at a potential cache-related bottleneck or a point of maximum cache efficiency, after which the cache can no longer significantly influence performance gains.
- Overall, these trends underscore the Pi's potential as an efficient single-task system and the Server's robustness in a multitasking environment.
- A notable point to be made is the fact that the Pi even with its Puny architecture, performs the best among all the other variants, making it the best option among the others for single-threaded tasks.

#### 6.1.1.2 LL (Last Level) Cache Misses

Digging deeper into the reason for the convergence, we look at the Last-Level Cache Misses, in Fig.6.2 to get an idea of whether our intuition of the cache's involvement in the time taken is correct.

- In the smaller memory allocations, the Raspberry Pi demonstrates more last-level cache misses than the Server in both concurrent and baseline scenarios, this could be attributed the smaller L2 cache size (1 MiB) which might get saturated more quickly than in the server.
- The Server, with a substantially larger 50 MiB L3 cache, initially shows fewer cache misses, which indicates a superior ability to cache more data and instructions, thereby reducing the need to fetch from slower main memory. this highlights the impact of cache on the performance particularly at the lower memories.
- The steady increase in cache misses for the Pi, even as memory allocation grows,

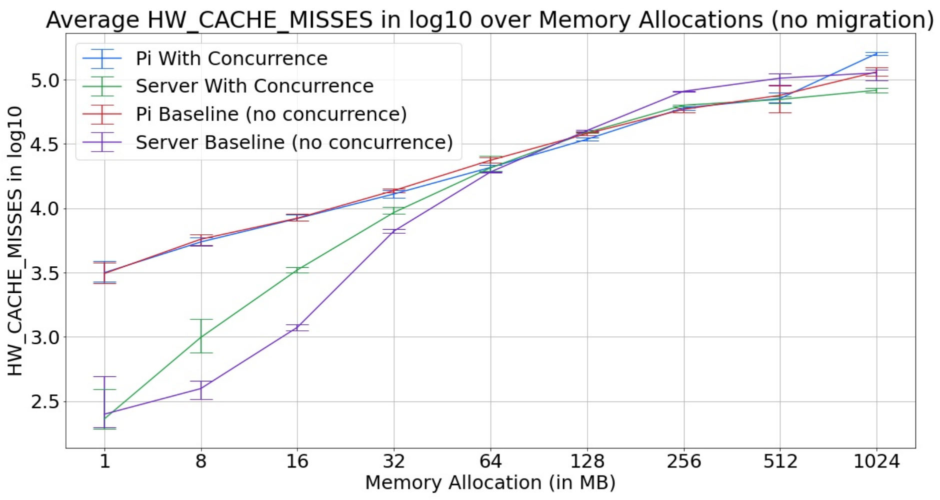


Figure 6.2: Figure showing the Average LLC Misses per Memory allocation along with the min and max variation

could be due to the fact that the Pi's L2 cache does not have enough space to benefit from larger memory allocations. Thus, the cache miss rate increases linearly with memory demand.

- The Server's drastic increase in cache misses at higher memory allocations, particularly for the concurrent operation, could be a result of cache contention among multiple processes. However, the large 50 MiB L3 cache appears to mitigate this effect until a certain point, beyond which the effect that the cache has does not result in a superior performance as opposed to the Pi.
- The convergence of cache miss trends at around 64MB across both systems may reflect a balancing point where the efficiency of cache utilisation is optimised relative to the available cache size and the memory demanded. Despite the large difference in cache size, this could also indicate that both systems' caches are optimised for similar operational scales. Additionally, at higher memory allocations, the narrow difference in last-level cache misses suggests that the Raspberry Pi's L2 cache and the Server's L3 cache both scale in performance.
- The Pi's L2 cache, despite being significantly smaller than the Server's L3 cache, exhibits a performance profile surprisingly comparable to the Server, especially at higher memory allocations. This parity suggests that the Pi's workload is well-accommodated within its cache capacity and that its cache efficiency is maintained even as memory demands increase.

#### 6.1.1.3 Branch Misses

Changing focus to the branch misses, from the Fig.6.3 see the following:

- Initially, the server demonstrates superior efficiency with fewer branch misses at the 1MB memory allocation mark. This suggests that for very small workloads, the Server's prediction algorithms or branch handling mechanisms might be more effective than the Pi's.

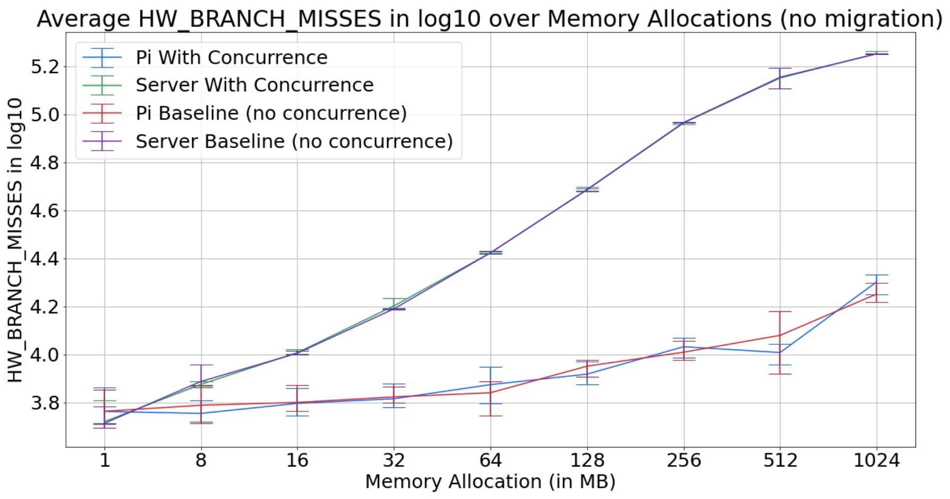


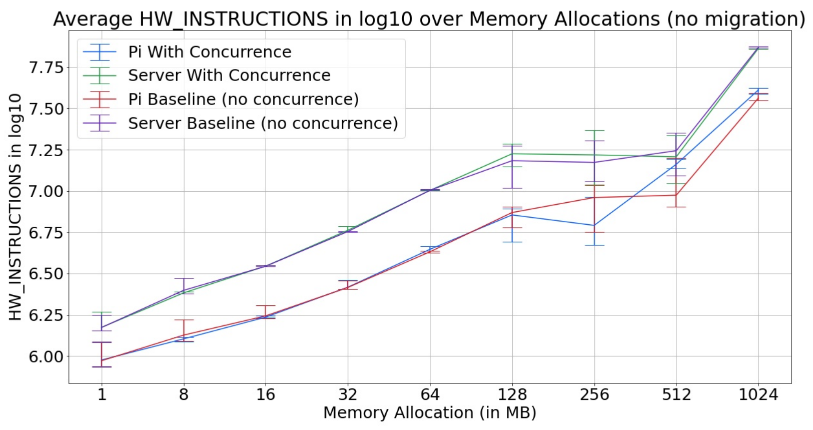
Figure 6.3: Average number of branch misses per Memory allocation along with the min and max variation

- As memory allocation increases beyond 1MB, the Pi consistently outperforms the Server with fewer branch misses. This could indicate that the Pi's architecture is more effective as the memory complexity increases, leading to a far smaller number of branch misses.
- The Server's branch misses rise drastically as memory allocations increase. This trend might reflect limitations in the Server's branch prediction capabilities at higher memory allocations, which may involve more complex branch decisions, resulting in more misses.
- Conversely, the Pi exhibits a much steadier and moderate increase in branch misses, reflecting a stable branch handling performance despite increased memory allocations.

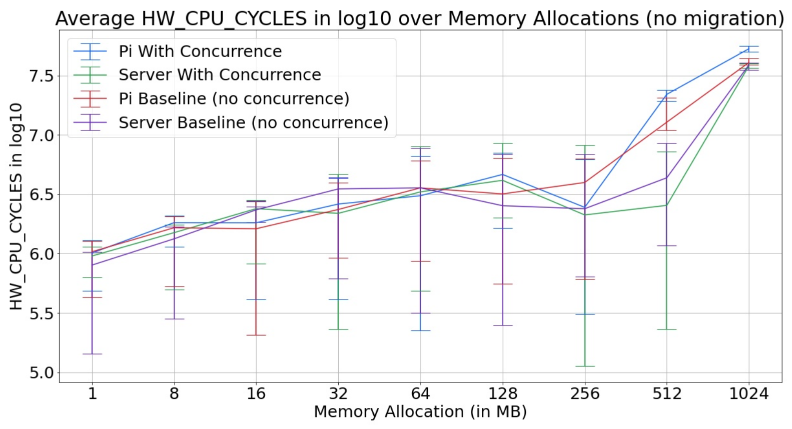
#### 6.1.1.4 Instructions and Cycles

Looking at the Instructions in the Fig.6.4a:

- Both the Raspberry Pi and Server show minimal differences between their baseline and concurrent executions across all memory allocations, suggesting that the presence of concurrent processes does not significantly impact the instruction throughput of either system.
- The Raspberry Pi consistently records lower instruction count than the Server by approximately 1.8 (0.25 in log10 terms) across memory allocations. This indicates that the Pi is processing fewer instructions overall, which could be interpreted as a more efficient execution for the given workload or less complex operations being performed.
- The gap in instruction counts is present even from the lowest memory allocation, highlighting the Pi's efficiency being less instruction-intensive on the Pi as compared to the Server.



(a) Instructions



(b) Cycles

Figure 6.4: Average Instructions and CPU cycles per Memory Allocation along with the min and max variation

Looking at the Cycles in Fig.6.4b:

- The number of CPU cycles remains relatively stable for both systems across different memory allocations and execution types, showing that the number of cycles required for processing does not fluctuate significantly with the introduction of concurrent processes or with changes in memory allocation.
- However, the Server with concurrent execution exhibits marginally better performance compared to the other scenarios in the higher memory scenarios (specially the 512MB), which could suggest slightly more efficient execution of processes in parallel.
- The Pi with concurrent execution appears marginally less efficient than the other scenarios. Although the difference is small, this might point to the Pi's limitations when handling multiple processes simultaneously.

Given that we see a variability in mostly the Instructions and the CPU Cycles, remain

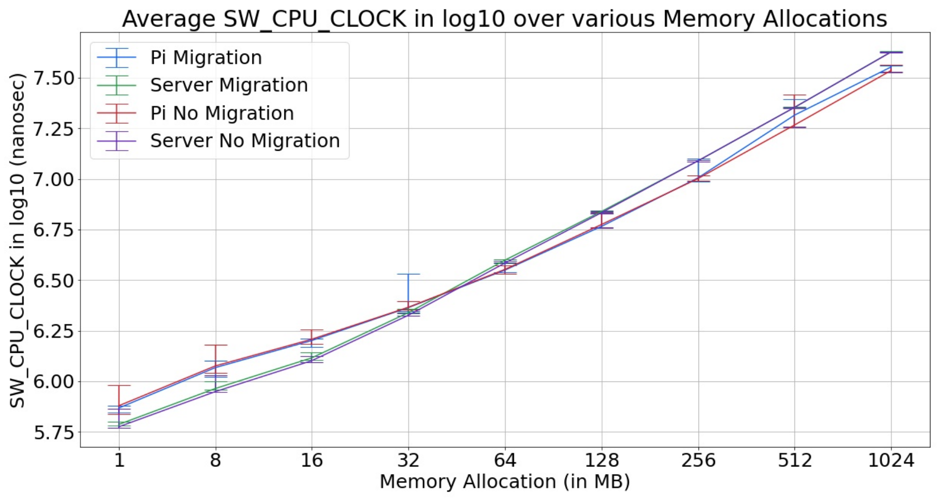


Figure 6.5: Average number of CPU clock time (in nanoseconds) per Memory allocation along with the min and max variation

relatively consistent between the Pi and the server, we can infer that the IPC (Instructions per Cycle) throughput for the Pi would be about 1.8 lower than the Server. The Server's higher IPC suggests it handles multi-threading and process management more effectively than the Pi.

### 6.1.2 With Migration Type

Comparing the impact of CPU migration with that of No Migration, helps us get insight into Resilience to Overhead. If a system maintains stable performance metrics with migration, it indicates resilience to the overhead that comes with moving processes between CPUs or cores. It is important to note that these graphs do not depict concurrent executions.

#### 6.1.2.1 CPU Clock

Again, starting our analysis in the CPU Clock in Fig.6.5

- The Raspberry Pi and the Server show increased CPU clock times when compared to their respective performances without migration. This increment highlights the overhead associated with CPU migration, yet it is interesting to note that both systems maintain a parallel performance trend, implying that the impact of migration is proportionally similar across the hardware architectures.
- Both systems exhibit a consistent increase in software CPU clock times as memory allocation grows, which is typical behavior reflecting the additional computational load associated with handling the larger memory.
- Although the Server started off performing better than the Pi, post (approximately) 32MB, the Pi consistently demonstrates superior performance.
- Contrary to the observations in 6.1.1.1, the convergence of performance, here, lies around the 32MB memory allocation for these plots. This suggests that the

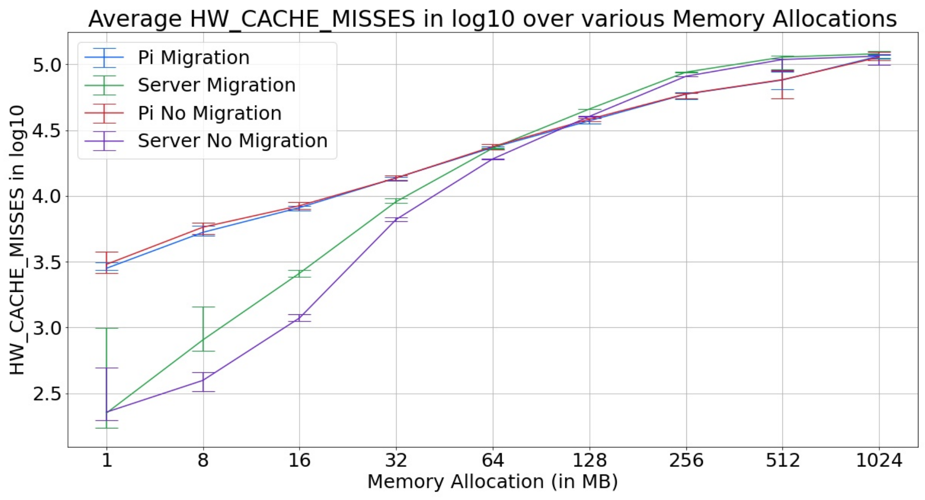


Figure 6.6: Figure showing the Average LLC Misses per Memory allocation along with the min and max variation

migration process might be more affected by the cache than the non-migration counterpart.

- The trends indicate that migration imposes a consistent overhead on both systems, but this does not disproportionately disadvantage either system. Even when subjected to the additional demands of migration, Pi's superior performance in single-threaded tasks is solidified.
- On the other hand, the Server, with its larger cache, doesn't show a drastic performance change between migration and no migration scenarios, reinforcing its capability to efficiently manage multitasking workloads across various memory allocations.

#### 6.1.2.2 Last Level Cache Misses

Similarly to the section 6.1.1.2, we dig deeper into the reason for the convergence, by looking at the Last-Level Cache Misses, in Fig.6.6 to validate our cache effectiveness assumption.

- Similarly to section 6.1.1.2, in the smaller memory allocations, the Raspberry Pi demonstrates more last-level cache misses than the Server in both migration scenarios, which too could be attributed the smaller L2 cache size.
- Similarly, the convergence of cache miss trends at around 64MB across both systems is also observed. Which too, may reflect a balancing point where the efficiency of cache utilisation is optimised relative to the available cache size and the memory demanded.
- In the Server, more cache misses are observed when migration is factored in. This is evident across all memory allocations up to 256 MB underscoring the impact of migration on cache performance. Beyond this allocation, cache misses plateau, suggesting a threshold where cache invalidation from migration becomes a consistent, manageable factor in performance.

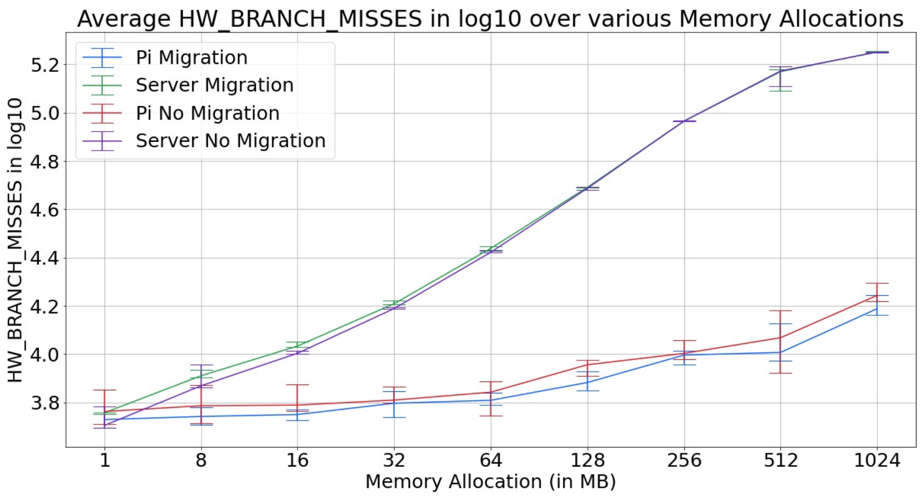


Figure 6.7: Average number of branch misses per Memory allocation along with the min and max variation

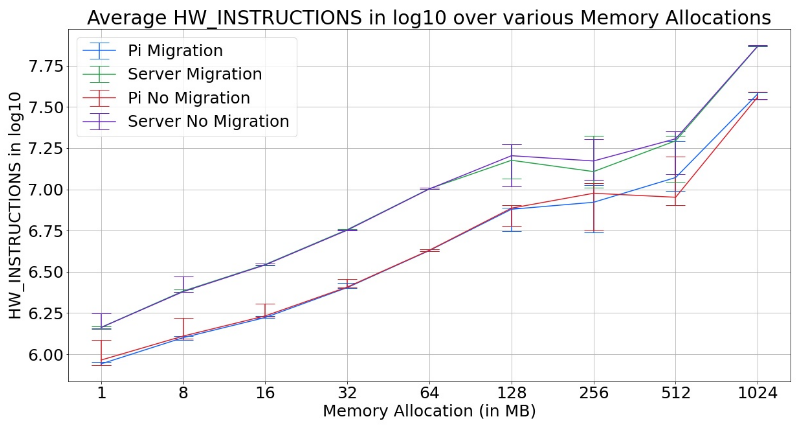
- The negligible difference in cache misses between the migration and no migration scenarios for the Pi indicates that CPU migration does not severely disrupt its cache's performance. This could point to an architecture that is less sensitive to the cache invalidation typically caused by migration processes.
- Overall, these observations reinforce the Pi's ability to maintain superior performance, highlighting its suitability for tasks that fit within its cache's constraints, without being heavily influenced by increased memory demands or the overhead of CPU migration.

#### 6.1.2.3 Branch Misses

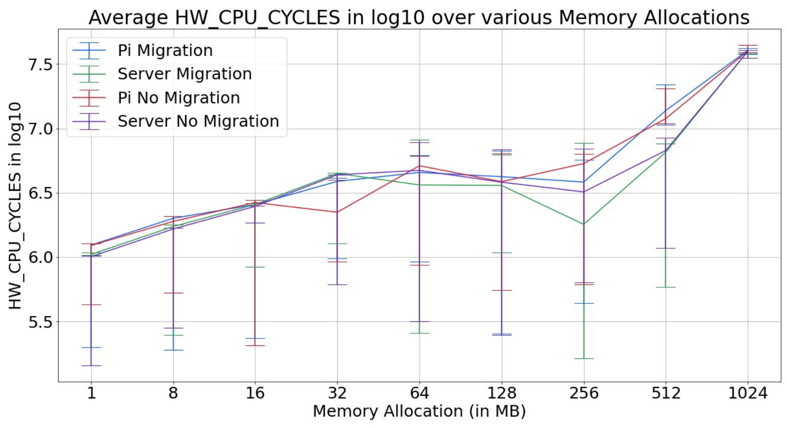
Further focusing on the branch misses from Fig.6.7, we observe:

- Much like the plot in 6.1.1.3, the overall trend of the server performing marginally better in 1MB memory allocation as compared to the Pi holds here as well. Likewise, the trend of the Pi surpassing the server's performance in other memory allocations is also maintained. The pattern showcasing a significant increase in the server is also upheld.
- The steady performance for the Pi implies an architectural resilience to CPU migration, maintaining branch prediction accuracy despite potential cache invalidation caused by migration.
- However, curiously, while in the server's case, the migration case performs slightly worse till the 128 MB, after which it converges with its non-migration counterpart, the Pi showcases a marginally better performance throughout in the migration case.
- The Pi's performance, exhibiting marginal improvement in the face of migration, suggests its processing is less disrupted by the additional overhead, possibly due to a more straightforward migration process that aligns well with its operational capabilities.

#### 6.1.2.4 Instructions and Cycles



(a) Instructions



(b) Cycles

Figure 6.8: Average Instructions and CPU cycles per Memory Allocation along with the min and max variation

Even though, there is not much of a difference between the figures 6.4a and 6.8a, we notice that the number of cycles at 1024MB for all the plots converge in fig.6.8b.

Therefore, the IPC calculated, would remain relatively the same for the Pi but increase for the Server due to a lower number of

## 6.2 Tracing Results

### 6.2.1 Overall Function Execution

To get a general idea of the function execution, a flamegraph such as the one in fig.6.9 was generated for both, the Pi and the Server. This lead to While, this helped get an intuitive idea of the parent function depth for the hotspot in the fork() syscall. This can be seen as the functions at depth 6 from the fig.6.9, this was also confirmed to be the case in the flamegraphs for all the memory allocations.

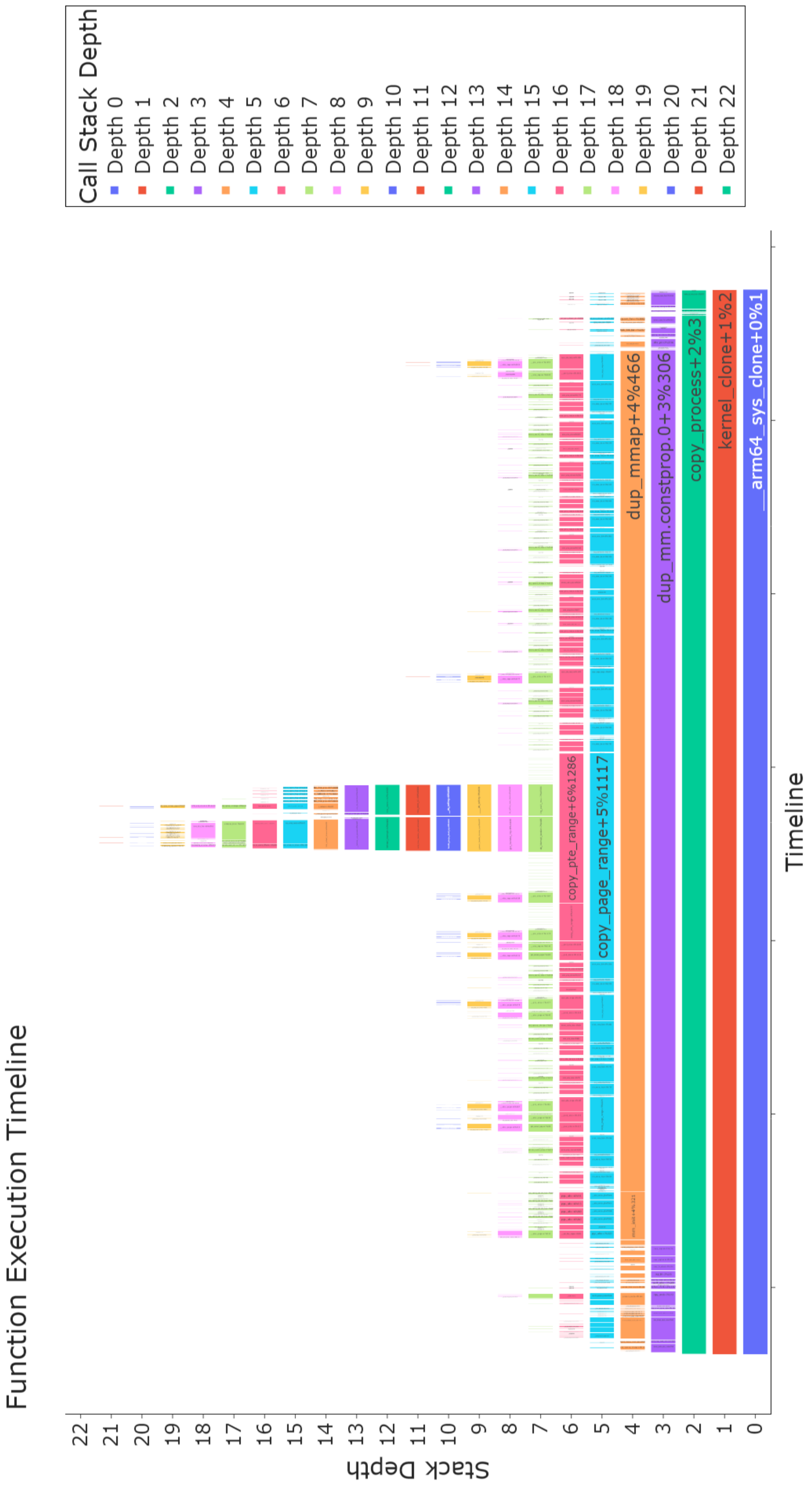
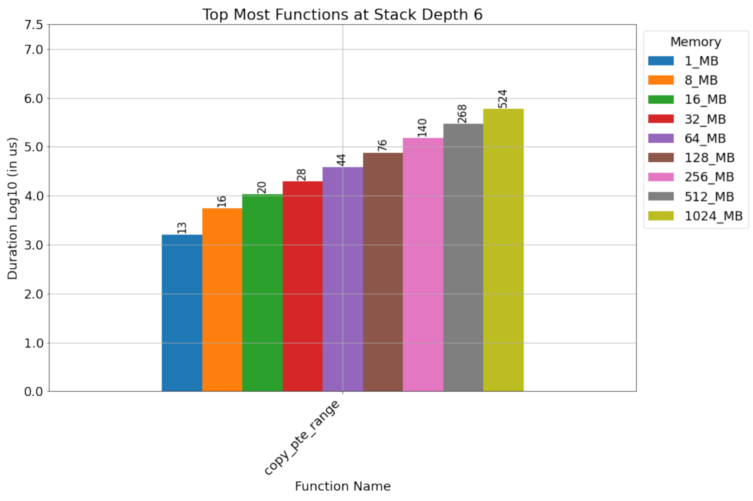


Figure 6.9: FlameGraph depicting the function execution for 1MB (in Pi)

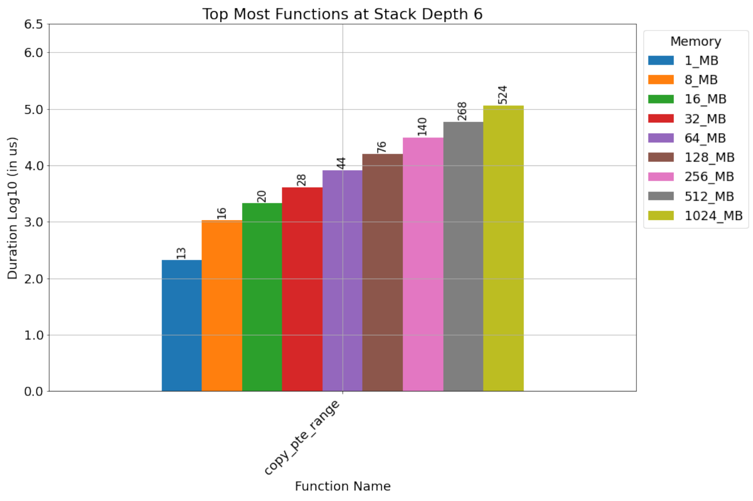
The only difference among the flamegraphs for the different memory allocations ended up being the number of ‘peaks’, like the one seen after ‘`copy_pte_range+6%1286`’, that can be seen starting mostly at the same function ‘`copy_pte_range`’. A few examples for these can be found in the Appendix A.

## 6.3 Hotspot Investigation - Single Process Execution

To understand the “Hotspot” we start by building upon the insight gained that the bulk of the function execution occurs starting at the stack depth 6, we plot using the method described in section 5.2.2.2. Comparing the Stack in the depth 6 for the server and the Pi, from the figure 6.10, we see that not only do the overall behaviour, increase in duration, between the pi and the server remain the same, but the number of times the functions gets called remain the same.



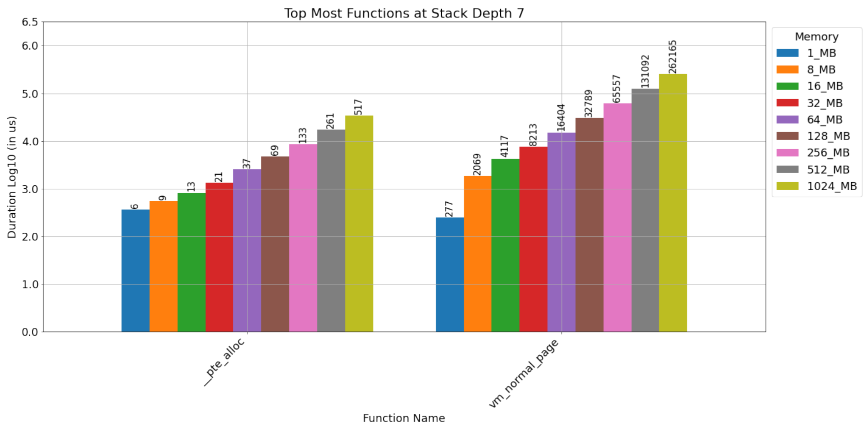
(a) Stack 6 Cleaned for Pi



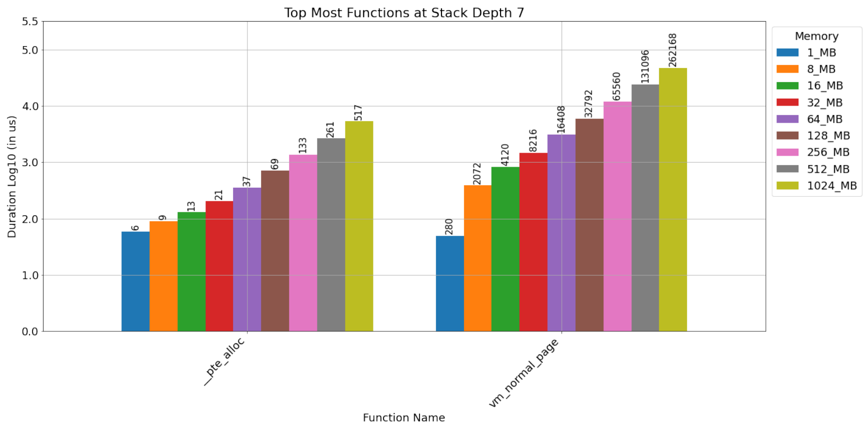
(b) Stack 6 Cleaned for Server

Figure 6.10: The filtered out function durations with the frequency of calls (annotated on bar) for Stack Depth 6

This indicates that up to stack level 6, the implementation of the fork() system call does not depend on the system architecture. Upon delving deeper into stack level 7 and excluding irrelevant interrupt function traces (since we are only concerned about the actual syscall implementation), we encounter the illustration in Figure 6.11. The filtered functions presented here showcase similar trends and almost identical function call counts. However, when examining the flame graphs, it becomes apparent that the **'vm_normal_page'** function doesn't invoke any other child functions. Since vm_normal_page is both, frequently called and has a non-trivial execution time is in comparison to other functions in the syscall, it **can be considered a architecturally-neutral hotspot**.



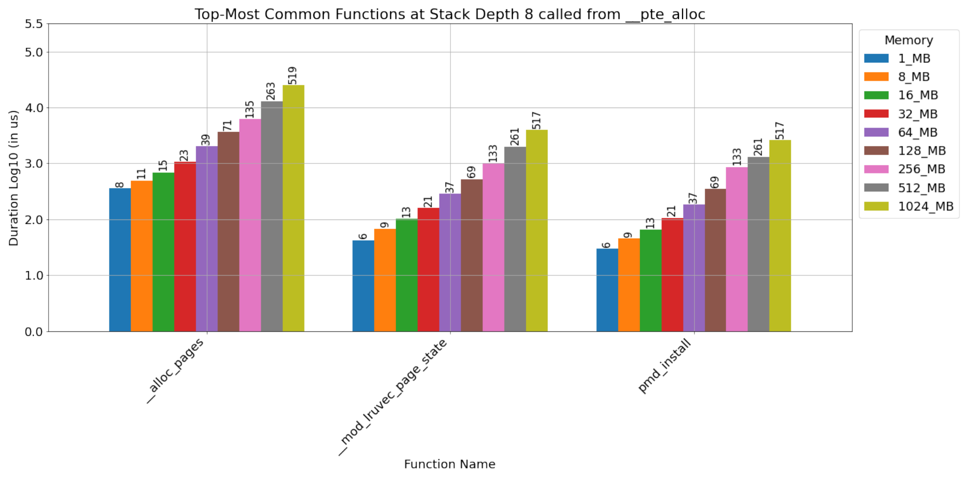
(a) Stack 7 Cleaned for Pi



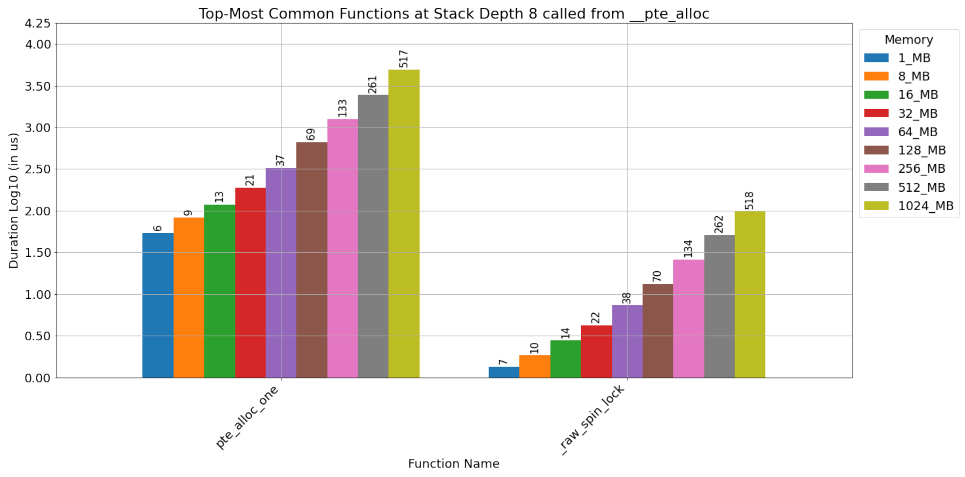
(b) Stack 7 Cleaned for Server

Figure 6.11: The filtered out function durations with the frequency of calls (annotated on bar) for Stack Depth 7

Since the 'vm_normal_page' does not call any functions, we direct our attention down the call stack for **'pte_alloc'**. Leading us to the figure 6.12. This shows us the first result for the pure architectural difference. While the server as shown in Fig.6.12b, depicts two functions, both of which are called a near identical number of times. The Pi shows three functions (Fig.6.12a, all of which, again, are called a near identical number of times. This shows that the 'pte_alloc' function has quite a different implementation in the Pi (ARM64 architecture) and the Server (x86 architecture). Furthermore, the fact that the number of calls for each of the filtered functions, for both the Pi and the Server, remain nearly identical. Therefore, giving the inference that the functions that are architecturally different in the highest level in the implementation of the fork() syscall for the ARM64 architecture are: **`'__alloc_pages'`**, **`'__mod_lruvec_page_state'`** and **`'pmd_install'`**.



(a) Stack 8 of functions called by '__pte_alloc' in Pi



(b) Stack 8 of functions called by '__pte_alloc' in the Server

Figure 6.12: The filtered out function durations with the frequency of calls (annotated on bar) for Stack Depth 8 called by '__pte_alloc'

While the pte_alloc_one is typically used for allocating a single page table entry. The function, in the Pi are described as doing:

1. **`__alloc_pages`**: This function performs the allocation of physical memory pages by checking the system's available memory and assigning the needed pages, ensuring efficient use of memory resources.
2. **`__mod_lruvec_page_state`**: This function adjusts the count of pages in a Least Recently Used (LRU) list within a specific memory group. Updating statistics by increasing or decreasing the count based on page allocations / de-

allocations, helping the system manage memory more effectively by keeping track of page usage patterns.

3. **`pmd_install`**: This function acquires a lock on the page middle directory to ensure that the PMD entry is vacant. In which case, it populates the entry with the provided page table, increments the system's page table entry count, and releases the lock.

Therefore, we can pin the differences in the Single Function execution between the Pi and the Server on these three functions.

## 6.4 Hotspot Analysis - Concurrent vs Single

Curiously, we observed that the concurrent execution of the fork() syscall in the Pi performed far worse than the baseline (single execution), to do so we first look at the to observe whether there is a different pattern that emerges.

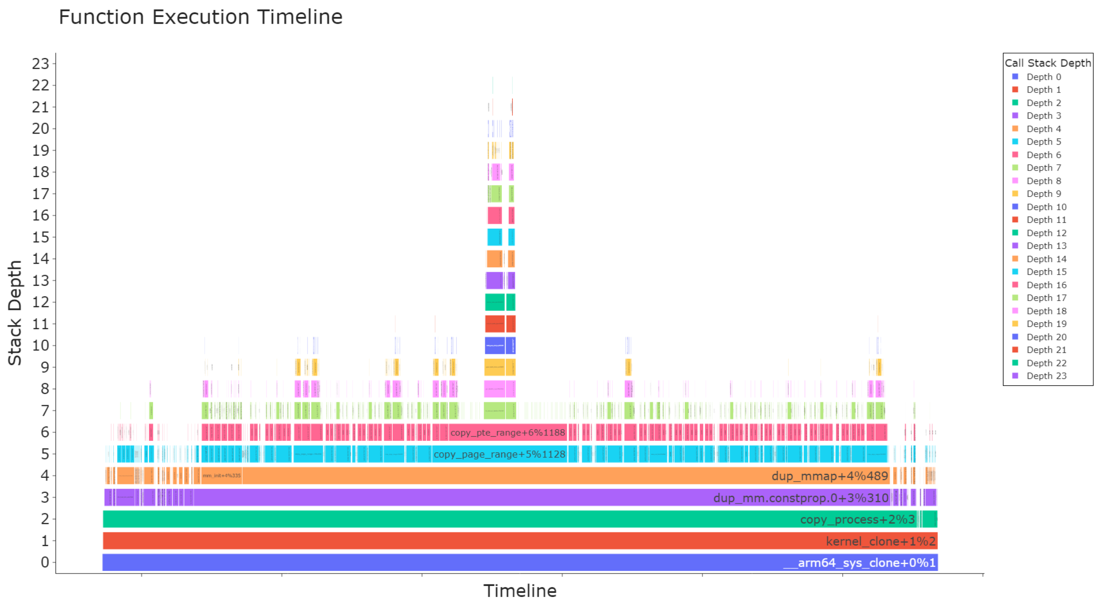


Figure 6.13: Flame Graph in concurrent execution of the workload for 1MB in the Pi

Since at a higher glance, this does not show much of a difference than the flamegraph for the single execution, we instead change our strategy to check the function at each depth to see how the execution differs. To do so, we devise a method to calculate the difference for each level subtracting the duration for the single execution from the concurrent execution.

While the stack depths at 0 to 6 did not show much of a difference, other than the fact that the total interrupts in the concurrent process was much more prominent as compared to the single execution, though their duration was not much. The stack depth 7 (table found in 6.1) showed a possible root cause for the added latency, the **`'vm_normal_page'`** function.

Digging deeper into the use of the function [Torvalds], we learn that the **`vm_normal_page`** function used within the virtual memory management subsystem to determine if a page table entry (PTE) points to a ”normal” memory page. Which means, it checks if the PTE corresponds to a standard, non-special memory page that can be directly manipulated or accessed.

Table 6.1: Difference between the Concurrent execution and Single Execution table

| | 1_MB | 8_MB | 16_MB | 32_MB | 64_MB | 128_MB | 256_MB | 512_MB | 1024_MB |
|---|---|---|---|---|---|---|---|---|---|
| vm_normal_page | 11.671 | 264.527 | 126.969 | 173.575 | 2339.596 | 5768.137 | 6118.552 | 6252.529 | 8357.569 |
| __rcu_read_lock | 2.81 | 0.244 | 3793.668 | 2.962 | 743.653 | 2.431 | 1.241 | 2.1 | 3.116 |
| mem_cgroup_kmem_disabled | 0.241 | 0.186 | 0.13 | 0.127 | 0.558 | 0.333 | 0.371 | 0.258 | 0.646 |
| __set_cpus_allowed_ptr_locked | 0.501 | 0.334 | 0.11 | 0.482 | 0.426 | 0.462 | 0.019 | 0.055 | 0.037 |
| __calc_delta | 0.02 | 0.111 | 0.298 | 1.445 | 1.058 | 1.223 | 1.538 | 1.37 | 0.833 |
| put_prev_entity | - | 72.481 | - | 71.76 | - | - | - | 33.221 | 70.0 |
| generic_handle_domain_irq | - | 9.852 | - | 178.741 | - | - | - | 187.037 | 565.649 |
| psi_group_change | - | 6.369 | - | 5.925 | - | - | - | 5.741 | 6.315 |
| set_next_entity | - | 4.408 | - | 4.834 | - | - | - | 4.352 | 4.611 |
| fpsimd_save | - | 2.093 | - | 1.926 | - | - | - | 1.982 | 1.963 |

Therefore, at concurrent execution, the function **‘vm_normal_page**’ might be causing an increase in CPU clock time and cycle counts due to the fact that it might indicate a heavier demand on the memory management of the operating system.

# Chapter 7

# Conclusions and Future Works

## 7.1 Performance Conclusion

Following the results we got in the Performance Experiments in section 6.1, we can safely conclude

1. Despite the Raspberry Pi's lower overall performance when handling the workload in comparison to the Server, its more favourable branch miss rates and gradual increase in cache misses (as compared to the server's) underscore its performance efficiency. This is particularly noteworthy given the Pi's smaller L2 cache (1MiB in the Pi and 5 MiB) and a complete lack of L3 cache. The Pi's lower software CPU clock times in baseline conditions reflect an architecture that, while computationally less powerful, is highly efficient for single execution tasks. The Server's higher instruction count, contrastingly, affirms its computational prowess.

2. The marginal advantage observed with no migration suggests that for workloads frequently employing fork() operations, CPU thread pinning can be highly beneficial for the Raspberry Pi, which tends to fare better without migration, highlighting the potential for optimisations around thread management in the ARM64 architecture. In contrast, the Server's performance remains consistent regardless of migration, demonstrating its robustness and capability to manage thread migration without significant performance trade-offs.

3. The influence of the Last Level Cache (LLC) on the workload and, by extension, the fork() syscall appears to be minimal. This indicates that for the evaluated workloads, the LLC does not play a critical role in influencing performance outcomes. This could point to efficient cache management that mitigates the potential negative impact of LLC on system performance.

4. The Raspberry Pi's CPU operates at a much lower clock speed compared to the Server's CPUs (1.50 GHz to 2.20GHz). Despite this, the Pi demonstrates a commendable level of efficiency as evidenced by its lower software CPU clock times in baseline scenarios. This efficiency becomes particularly pronounced when considering the fact that the Pi, operating at a lower frequency, still man-

ages to achieve comparable performance in certain non-computational metrics such as cache misses and a much better performance in branch miss predictions. This reinforces the idea that clock speed is not the sole determinant of performance; architectural optimisations and workload characteristics are equally vital in achieving efficient system performance.

5. Meanwhile, the Server's significantly better CPU clock speed corresponds with its superior computational power, evident from its higher rate of processed instructions. It is also evident that the Server is engineered to deliver steady performance through a wide spectrum of tasks, suggesting a harmonised relationship between its raw speed, multi-threading efficiency, and adept process management.

6. These findings suggest that the traditional Server's edge over the Raspberry Pi is predominantly rooted in its superior computational capabilities rather than cache or branch related performance.

## 7.2 Hotspot Conclusion

Due to the slightly peculiar behaviour experienced in the Raspberry Pi, highlighting the architectural differences in the highest-level implementation of the fork() syscall between the ARM and the x86 architectures, our investigation goes beyond just performance comparison between the Pi and the Server. We aim to elucidate these distinctions not only in terms of syscall execution but also to serve as a beacon for understanding how the architectural nuances influences the fork behaviour across the platforms. This analysis intends to bridge the gap between theoretical architecture and practical application, offering insights that can guide optimisation strategies and enhance system design for tailored performance across diverse hardware ecosystems.
These findings highlight the Raspberry Pi's ability of managing single-threaded tasks compared to the Server and the Pi's multitasking bottlenecks, attributable in part to differences in memory management strategies, cache utilisation, and the handling of virtual memory pages. The analysis not only deciphers the underpinnings of architecture-specific performance traits but also sets the stage for further exploration into optimising syscall implementations across diverse hardware platforms.

## 7.3 Future Works

The exploration undertaken in this study has shed light on the intricate relation of the fork() system call performance across the ARM and x86 architecture, particularly highlighting areas where the Raspberry Pi and Server architectures diverge in efficiency. While our findings provide valuable insights into the complexities of concurrent and single-process executions, they also open the door to a plethora of questions regarding optimisation, scalability, and architectural adaptability. Recognising the constraints of our current scope and the ever-evolving landscape of computing architectures, the subsequent pathways for research present an exciting ground for enhancing performance across systems. In this vein, the Future Works section outlines several directions for forthcoming investigations, aimed at not only addressing the limitations encountered but also expanding the horizon of what is currently understood about system call optimisation and architectural efficiency.

1. Investigate optimisation techniques that can mitigate the observed performance bottlenecks in concurrent executions, especially focusing on memory management strategies that can be adapted across different architectures like ARM and x86. This could include comparing various novel and theoretical Copy-on-write techniques, such as the On-demand Fork[Zhao et al., 2021], Async-Fork [Pang et al., 2023], and Other Copy-on-Write techniques such as "CCoW: Optimizing Copy-on-Write Considering the Spatial Locality in Workloads" [Ha and Kim]

2. Since this project was undertaken before the Raspberry Pi 5 was generally available, performing further analyses to compare the effectiveness of the newly included L3 cache to the Pi could be a significant insight into whether the Pis could be placed closer to the edge to successfully"Democratize" it without leaving any performance out of the table.

3. Given the significant role that the cache behaviour played in influencing performance, a detailed study focusing on cache management strategies, including cache prefetching, cache locking, and cache partitioning, could yield insights into improving performance across different memory allocation scenarios.

4. Although this thesis dealt utilised a simulation of workloads that would be fork() intensive, an extension to this study could include real-world applications that are typical for edge computing tasks, such as Redis, Fuzzers, etc., as a point of comparison between the Raspberry Pi and the Server.

# Bibliography

Redis latency monitoring. URL https://redis.io/docs/management/optimization/latency-m

ARM® Cortex®-A72 MPCore Processor Technical Reference Manual, Revision: r0p3, 2016. URL https://developer.arm.com/documentation/100095/0003/?lang=en. Copyright © 2014-2016, ARM Limited or its affiliates. All rights reserved.

Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342407. doi: 10.1145/2901318.2901350. URL https://doi.org/10.1145/2901318.2901350.

Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 14–22, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367271. doi: 10.1145/3317550.3321435. URL https://doi.org/10.1145/3317550.3321435.

Tim Bird. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*, volume 1. Citeseer, 2009.

Francisco Carpio, Marc Michalke, and Admela Jukan. Benchfaas: Benchmarking serverless functions in an edge computing network testbed. *IEEE Network*, 37(5): 81–88, 2023. doi: 10.1109/MNET.125.2200294.

Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL https://www.flux.utah.edu/paper/duplyakin-atc19.

Brendan Gregg. CPU flame Graphs. URL https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

Minjong Ha and Sang-Hoon Kim. CCoW: Optimizing copy-on-write considering the spatial locality in workloads. 11(3). ISSN 2079-9292. doi: 10.3390/electronics11030461. URL https://www.mdpi.com/2079-9292/11/3/461.

Anurag P. Jadhav and V.B. Malode. Raspberry pi based offline media server. In *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pages 531–533, 2019. doi: 10.1109/ICCMC.2019.8819718.

Linux Man-pages Project. perf_event_open(2) - Linux man page. `https://man7.org/linux/man-pages/man2/perf_event_open.2.html`, 2024. Accessed: 2024-03-31.

Pu Pang, Gang Deng, Kaihao Bai, Quan Chen, Shixuan Sun, Bo Liu, Yu Xu, Hongbo Yao, Zhengheng Wang, Xiyu Wang, Zheng Liu, Zhuo Song, Yong Yang, Tao Ma, and Minyi Guo. Async-fork: Mitigating query latency spikes incurred by the fork-based snapshot mechanism from the os level, 2023.

Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. Democratizing the network edge. *SIGCOMM Comput. Commun. Rev.*, 49(2): 31–36, may 2019. ISSN 0146-4833. doi: 10.1145/3336937.3336942. URL `https://doi.org/10.1145/3336937.3336942`.

Charles Severance. Eben upton: Raspberry pi. *Computer*, 46(10):14–16, 2013. doi: 10.1109/MC.2013.349.

Linus Torvalds. GitHub - torvalds/linux: Linux kernel source tree. URL `https://github.com/torvalds/linux?tab=readme-ov-file`.

Xunzheng Zhang, Haixia Zhang, and Dongfeng Yuan. A platform base on rpeccf: Raspberry pi edge-cloud collaboration framework. In *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, 2020. doi: 10.1109/PIMRC48278.2020.9217253.

Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 540–555, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456258. URL `https://doi.org/10.1145/3447786.3456258`.

# Appendix A

# FlameGraphs

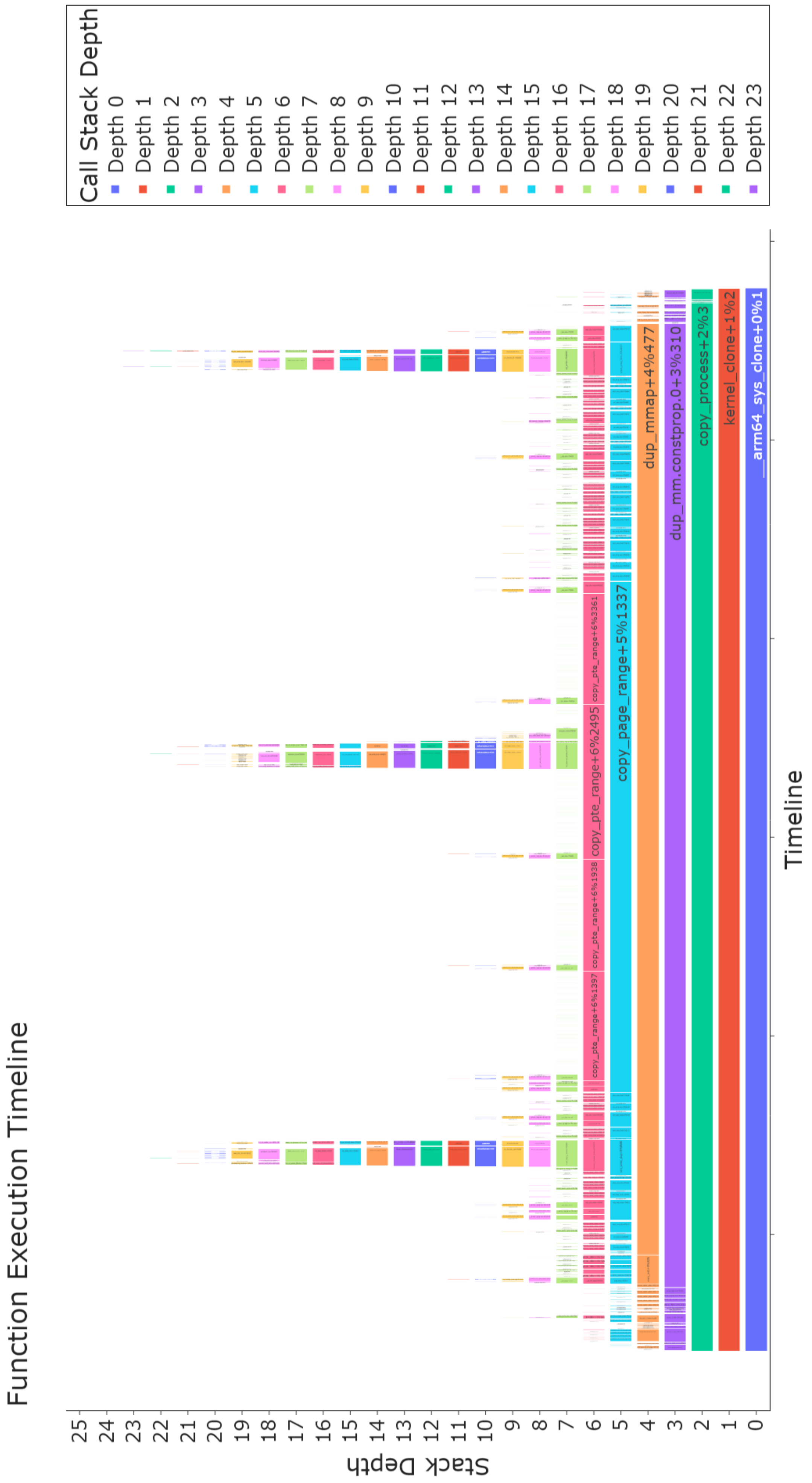


Figure A.1: FlameGraph depicting the function execution for 8MB (in Pi)

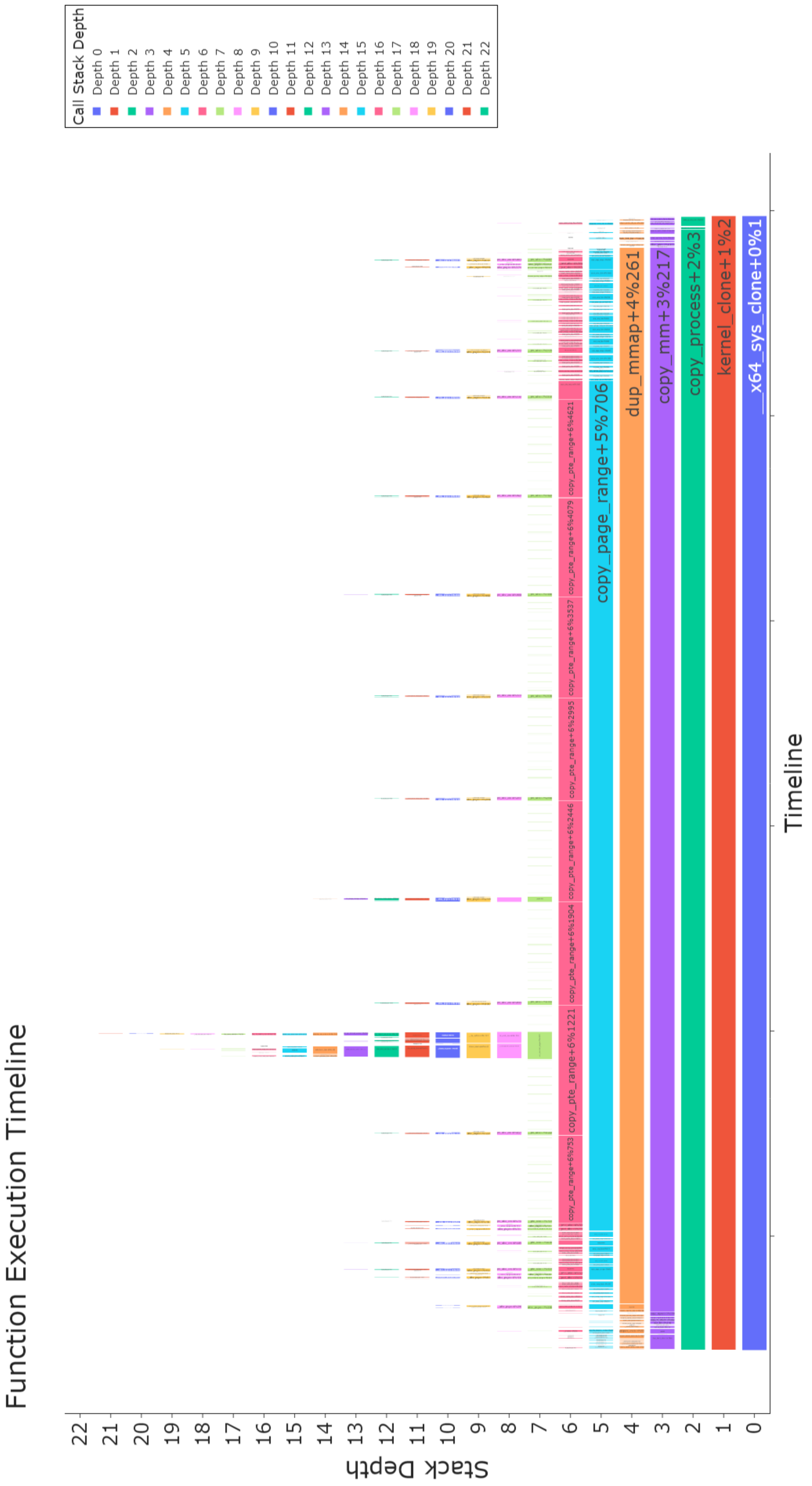


Figure A.2: FlameGraph depicting the function execution for 16MB (in Server)