School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Evaluating Programming Languages for Verified LionsOS Components

by

Zhewen Shen

Thesis submitted as a requirement for the degree of

Bachelor of Advanced Computer Science (Honours)

Submitted: December 2025

Supervisor: Prof. Gernot Heiser Student ID: z5388136

Abstract

Achieving end-to-end verification of operating system components requires removing the compiler from the trusted computing base. Pancake is a systems programming language with a verified compiler, designed for low level code such as device drivers. This thesis evaluates Pancake's suitability for implementing components of LionsOS, a modular operating system built on the seL4 microkernel.

We port over 20 LionsOS components to Pancake, comprising over 5,500 lines of code, including device drivers, virtualisers, and the core runtime library. We document the engineering effort involved and measure performance against C compiled with Clang and the verified CompCert compiler. Through targeted optimisations, we bring Pancake components within 10 to 15 percent overhead of their C counterparts. We conclude that Pancake is a viable language for building verification ready LionsOS components.

Acknowledgements

I would like to thank my supervisor Prof. Gernot Heiser for his guidance, feedback, and patience throughout this project. I also thank my assessor Dr. Peter Chubb for his insightful comments and suggestions.

I thank Ivan Velickovic for his help debugging numerous issues and for improving my understanding of the systems we work with. His willingness to answer my many questions made the debugging process far less painful. I also thank Courtney Darville and Krishnan Winter for their support with benchmarking setup and for helping me navigate the benchmarking results. I thank Alex Long for keeping the boards I needed for benchmarking alive.

I thank the Pancake team for their continuous work this year and for being responsive to issues I encountered. In particular, I thank Junming Zhao for helping me get started with Pancake and for providing existing work to build upon. I also thank Halogen Truong, Miki Tanaka, and Minh Do for their technical support with Pancake throughout the project.

I thank my fellow thesis students, especially James Treloar, Michael Mospan, Etkin Tetik, and Chirag Sawlani, for the shared struggles, late nights, and moral support that made this year more bearable.

I thank the broader Trustworthy Systems group for their support and for creating an environment where I always felt welcome.

Lastly, I thank my family and friends for everything else.

Abbreviations

AADL Architecture Analysis and Design Language

AST Abstract Syntax Tree

CAmkES Component Architecture for microkernel-based Embedded Systems

CC Communication Channel

CPU Central Processing Unit

CVE Common Vulnerabilities and Exposures

DMA Direct Memory Access

DWMAC DesignWare Media Access Controller

ELF Executable and Linkable Format

FFI Foreign Function Interface

GCC GNU Compiler Collection

HOL Higher-Order Logic

I/O Input/Output

I^2C Inter-Integrated Circuit

IP Internet Protocol

IPC Inter-Process Communication

ISA Instruction Set Architecture

ITP Interactive Theorem Proving

KISS Keep It Simple, Stupid

LLVM Low Level Virtual Machine

LoC Lines of Code

lwIP Lightweight IP

MAC Media Access Control

NIC Network Interface Controller

OS Operating System

PD Protection Domain

PMU Performance Monitoring Unit

POSIX Portable Operating System Interface

PPC Protected Procedure Call

PSOS Provably Secure Operating System

RAM Random Access Memory

RTT Round Trip Time

RX Receive

SAT Boolean Satisfiability

SDK Software Development Kit

sDDF seL4 Device Driver Framework

SMB Server Message Block

SML Standard ML

SMP Symmetric Multiprocessing

SMT Satisfiability Modulo Theories

SoC System on Chip

TCB Trusted Computing Base

TLB Translation Lookaside Buffer

TX Transmit

UB Undefined Behaviour

UDP User Datagram Protocol

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In May 2017, the **WannaCry** ransomware outbreak dramatically underscored the fragility of modern operating systems. By exploiting an unpatched vulnerability in the Windows SMB protocol, WannaCry rapidly infected more than **230,000 computers across 150 countries**, crippling hospitals, businesses, and government services (National Institute of Standards and Technology, 2017). This was not an isolated incident; it highlighted a broader trend of escalating security failures in widely used operating systems. Linux, Windows, and macOS — the foundational platforms for servers, desktops, and mobile devices — have each been plagued by hundreds, even thousands, of new vulnerabilities every year. In 2024 alone, Microsoft Windows 10/11 (and various other server editions) had 4939 reported CVE-listed vulnerabilities, the mainline Linux kernel had 3889, and Apple's macOS had 519 (stack.watch, 2024). Some notable examples in recent years include "PrintNightmare"[1], a critical remote code execution vulnerability in the Windows Print Spooler service; "Dirty Pipe"[2], a privilege escalation flaw in the Linux kernel; and "Achilles"[3], a Gatekeeper bypass in macOS that allowed untrusted apps to execute without user consent. Many of these flaws are critical, enabling remote code execution or privilege escalation by attackers. The steady drumbeat of patches and security updates for all major OSes attests to an uncomfortable reality: **modern operating systems remain fundamentally insecure despite their importance**.

While this is tolerable in general-purpose computing, it is unacceptable in safety- and security-critical systems, such as those used in aircraft, medical devices, autonomous vehicles, and critical infrastructure, where a single failure can lead to catastrophic outcomes (Butler and Finelli, 1993; Rushby, 1997). These systems demand far stronger guarantees of correctness, isolation, and integrity than traditional OS architectures were ever designed to provide. In such domains, the operating system must not merely be

[1] `https://nvd.nist.gov/vuln/detail/CVE-2021-34527`

[2] `https://nvd.nist.gov/vuln/detail/CVE-2022-0847`

[3] `https://nvd.nist.gov/vuln/detail/CVE-2022-42821`

"secure enough"; it must be provably correct.

This requirement has motivated decades of research into **Provably Secure Operating Systems (PSOS)** (Neumann et al., 1975; Feiertag and Neumann, 1979; Neumann et al., 1980; Neumann and Feiertag, 2003) — systems for which critical properties can be mathematically verified. However, attempts to build such systems were historically hindered by impracticality or excessive cost and complexity in the engineering and verification effort. That changed with the seL4 microkernel (Klein et al., 2009), the first operating system kernel to be fully mathematically verified. seL4 provides strong spatial and temporal isolation, fine-grained access control via capabilities, and proofs of integrity and confidentiality. The seL4 project demonstrated that formal verification can be applied to real-world systems of significant complexity, and has since seen deployment in autonomous aircraft (Cofer et al., 2018) and autonomous vehicles (Qu, 2024).

Despite these successes, more than a decade after seL4's verification, a complete operating system stack built on top of it that is itself formally verified remains elusive. The challenges lie in both the **engineering** and **verification** of the components above the kernel. To address the engineering challenges, the seL4 Microkit (seL4 Foundation, 2023) and LionsOS (Heiser et al., 2025) were developed, demonstrating that performant and practical systems can be built on seL4 without significant overhead. However, verification beyond the kernel remains an open problem. To achieve end-to-end verification, components within the Trusted Computing Base (TCB) must be formally verified down to the executable code. In seL4's verification story, the C compiler was removed from the TCB through translation validation (Sewell et al., 2013), a fragile and laborious process that imposes limitations on code complexity. A more robust approach is to use programming languages with well-defined semantics and a verified compiler that produces *correct* binaries by construction.

Pancake (Pohjola et al., 2023) is a low-level systems language designed for this purpose. It builds upon the verified CakeML compiler backend (Kumar et al., 2014), providing a path to verified compilation for systems code. However, adopting a new language for systems development introduces practical concerns: how much engineering effort does porting require, what language limitations affect development, and what performance overhead does verified compilation introduce?

This thesis evaluates Pancake as a language for building LionsOS components. We port several core components to Pancake and provide both qualitative and quantitative analysis of the experience. On the qualitative side, we examine the engineering overhead introduced by Pancake's language limitations and discuss the practical challenges of the porting process. On the quantitative side, we measure the performance of Pancake components against both the original C implementations and CompCert (Leroy, 2009) compiled versions. CompCert is a verified optimising compiler for C, serving as our baseline for verified compilation performance. Through this evaluation, we assess whether Pancake is a practical choice for building LionsOS components without unacceptable compromises to performance or development effort.

Problem Statement

The aim of this thesis is to evaluate the suitability of Pancake as a systems programming language for implementing LionsOS components. To answer this, we make the following contributions:

1. We port the seL4 Device Driver Framework (sDDF) and `libmicrokit`, which together provide the core OS services for LionsOS. These ports provide a foundation for future LionsOS verification efforts.
2. We document the engineering effort for each port, measuring lines of code, binary size, and development time. We report on the practical challenges of using Pancake, including compiler limitations and debugging difficulties.
3. We systematically compare Pancake, Clang, and CompCert implementations of a complete sDDF networking stack, measuring throughput, latency, and CPU utilisation under realistic workloads.
4. We identify performance bottlenecks and implement targeted optimisations that bring our Pancake components within 10 to 15 percent of their C counterparts, demonstrating that Pancake is viable for performance critical systems code.

Chapter 2

Background

2.1 What is Formal Verification?

So far, we have been discussing verified and provably secure systems – but what does that actually mean? Given the scope of this thesis, we will not cover the technical details of it, but at a high level, formal verification refers to the use of mathematical proofs to ensure that a system satisfies its specification under all possible conditions. Unlike testing, which checks only specific cases, formal verification provides strong guarantees of correctness across all behaviours. As the seL4 verification effort notes, "complete formal verification is the only known way to guarantee that a system is free of programming errors"(Klein et al., 2010).

2.2 seL4 Microkernel: A Foundation for Verified Systems

This section provides an overview of microkernel architecture as a foundation for building high-assurance systems. We begin by discussing the core principles and evolution of microkernel design, followed by an overview of seL4 (Klein et al., 2009) as a representative third-generation microkernel. Finally, we outline the engineering challenges that arise when building a general-purpose operating systems on top of seL4.

2.2.1 What is A Microkernel?

The *kernel* is the core part of an operating system responsible for managing hardware resources and enforcing isolation between processes. It operates with the highest privileges and typically handles tasks such as memory management, CPU scheduling, and communication between software components.

A *microkernel* is a minimalist approach to operating system kernel design. Unlike a monolithic design, which places all core operating system services – such as device drivers, file systems, and networking – within the kernel, a microkernel includes only the most essential components, typically low-level address space management, thread scheduling, and inter-process communication (IPC) (Liedtke, 1995).

The remaining OS functionality runs in user space as separate, isolated services that communicate with each other via the IPC mechanism (Liedtke, 1995). This separation improves modularity, security, and stability, as faults in individual services are less likely to compromise the entire system. However, typical microkernel-based systems often faced performance challenges due to the overhead of IPC and context switching (Chen et al., 2024).

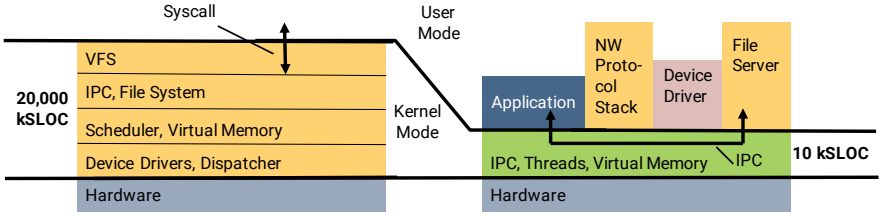

Figure 2.1: Monolithic (left) vs. Microkernel-based (right) Operating Systems (Heiser, 2020)

Evolution of microkernel design

Microkernel architectures have evolved over time in response to early limitations and shifting system requirements. Figure 2.2 illustrates the three major generations of microkernel design (Heiser, 2024). First-generation microkernels retained much of the functionality typical of monolithic kernels, resulting in many system calls and extremely poor IPC performance. In response, second-generation microkernels embraced minimality as a core principle, stripping unnecessary functionality from the kernel to achieve greater simplicity and significantly improved performance. Third-generation microkernels, such as seL4, further refined this approach by focusing not only on performance but also on formal assurance, including functional correctness and strong isolation guarantees. They achieve this by aggressively minimising the kernel and delegating responsibilities such as memory management to user space.

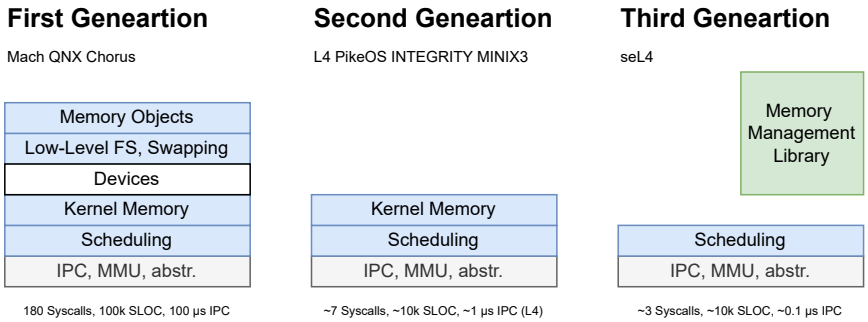

Figure 2.2: Generations of Microkernels. Adapted from (Heiser, 2024)

2.2.2 The seL4 Microkernel

The seL4 microkernel is a third-generation microkernel in the L4 family (Figure 2.3), tracing its lineage to Liedtke's work on high-performance kernels in the mid 1990s (Liedtke, 1995). Similar to its predecessors, seL4 adopts the minimality principle and provides only the minimal required mechanisms to securely abstract hardware (Heiser et al., 2022).

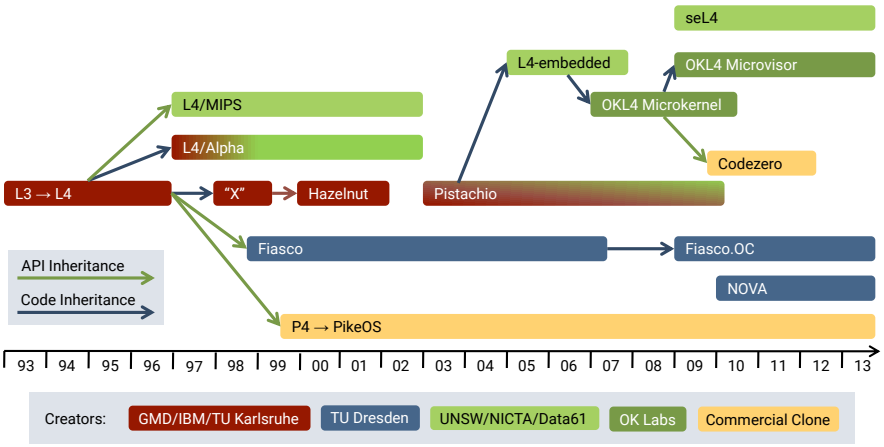

Figure 2.3: L4 microkernel family tree (Heiser et al., 2025).

What distinguishes seL4 is the depth of its formal verification. It was the first general-purpose operating system kernel to have a machine-checked proof of functional correctness down to the executable binary (Klein et al., 2014). This level of assurance ensures that the kernel's implementation precisely adheres to its specification, ruling out entire classes of bugs.

One of seL4's key design features is its use of a *capability-based access control* model, a concept dating back to early work by Dennis and Van Horn (1966). In this model, a capability is an unforgeable token that confers specific access rights to a kernel-managed

object, such as memory regions, threads, or communication endpoints. Rather than relying on global namespaces or privileged system calls, all access in seL4 is mediated through these capabilities, which are explicitly granted and transferred between components. This fine-grained model enforces strong isolation and explicit authority, enabling formal reasoning about system security properties.

Despite the additional proof infrastructure, studies have established seL4 as the world's fastest microkernel with performance within 25% of the limits imposed by hardware it runs on (Mi et al., 2019). The performance and assurance provided by seL4 makes it an attractive foundation for safety- and security-critical systems, such as autonomous vehicles, medical devices, and industrial control systems. Real-world deployments have demonstrated its potential, including use in autonomous military aircraft (Cofer et al., 2018) and commercial electric vehicles (Qu, 2024). However, such adoptions remain relatively rare and confined to highly specialised domains, reflecting the considerable engineering challenges associated with building practical and general-purpose systems directly on top of seL4's low-level API.

Challenges beyond the kernel

seL4's minimal design exposes developers directly to low-level mechanisms that are typically abstracted away in conventional operating systems. Core responsibilities such as memory management, including page table construction and management of kernel memory regions, are delegated entirely to user space (Elkaduwe et al., 2008). Although this delegation is critical for enabling provable isolation, it significantly increases the complexity of system construction. Developers must not only reason carefully about the allocation and protection of resources, but must also implement substantial infrastructure before a functional system can be developed.

As a result, seL4 has been described as the *"assembly language of operating systems"* (Heiser et al., 2025) – a microkernel that offers fine-grained and powerful primitives, but requiring a rather deep expertise to use correctly. For most developers, this level of complexity presents a substantial barrier to adoption. Projects that attempt to deploy seL4 at scale without substantial framework support frequently encounter unsustainable engineering overheads, and several early seL4-based initiatives were abandoned after a few years due to these challenges (Heiser et al., 2025). This has motivated several initiatives aiming to address the practical barriers of building systems on seL4, including CAmkES (Kuz et al., 2007) (Section 2.3.1), Microkit(seL4 Foundation, 2023) (Section 2.3.2), and LionsOS (Heiser et al., 2025) (Section 2.3.3).

2.3 Building Verified Systems on seL4

Thus far, we have provided a high-level overview of what a microkernel-based operating system looks like, and discussed the challenges involved in building practical systems

directly on seL4. In this section, we introduce two key frameworks, CAmkES and Microkit, which aim to reduce the engineering effort required to construct systems on top of seL4. Finally, we present LionsOS, a state-of-the-art operating system built using Microkit, designed to balance security, performance, and practicality.

2.3.1 Component Architecture for microkernel-based Embedded Systems (CAmkES)

CAmkES is a software development and runtime framework designed to simplify the construction of complex multiserver systems built on top of the seL4 microkernel. It follows a component-based engineering approach (see Figure 2.4), where a system is modelled as a set of isolated software components with explicitly defined interaction interfaces and connections (Kuz et al., 2007; seL4 Foundation, 2025).

CAmkES provides a language for describing the component interfaces, components themselves, and the overall system architecture. A build tool processes these descriptions and automatically combines developer-provided component code with generated scaffolding and glue code to produce a complete, bootable system image.

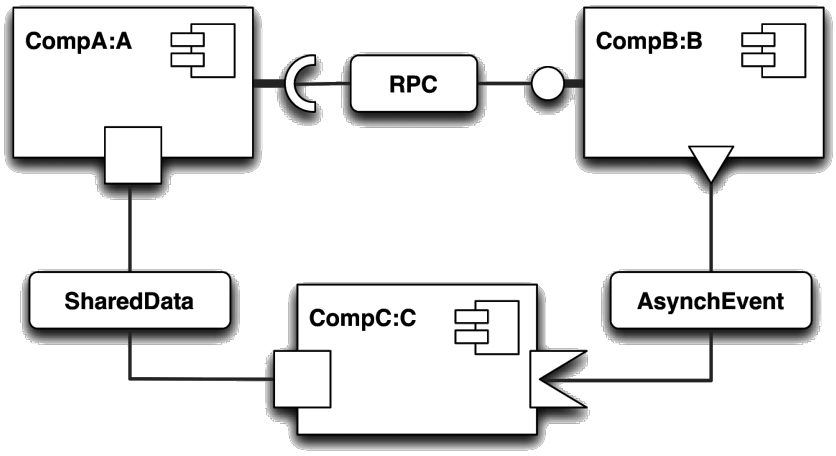

Figure 2.4: Basic CAmkES Architecture (Trustworthy Systems, 2025a)

However, CAmkES's abstractions come at a cost. Its component model introduces additional complexity, and its lack of a dedicated software development kit (SDK) results in a rather complex build process that is tightly coupled to the already complex kernel build system. Furthermore, the generated scaffolding introduces notable runtime overheads (Heiser, 2020), which can be problematic for performance-sensitive embedded systems. These limitations have motivated the development of more lightweight alternatives, such as Microkit, which aim to simplify system construction while preserving seL4's strong assurance and performance.

2.3.2 The Microkit Framework

The Microkit framework is a recent successor to the CAmkES framework, retaining its modular, component-based architecture while significantly reducing complexity. It is designed as a lightweight SDK for constructing statically architectured systems on seL4, where all components and their interactions are fixed at system build time. By exposing a minimal set of abstractions that closely mirror seL4's core mechanisms, Microkit makes it easier to correctly build systems on top of seL4 without incurring significant engineering or performance overheads (Trustworthy Systems, 2025b; seL4 Foundation, 2023).

The central abstraction in Microkit is the protection domain (PD), which is analogous to a process in traditional Unix-like systems. Each PD encapsulates an address space (VSpace), a capability space (CSpace), a thread and its scheduling context, along with basic synchronisation primitives. PDs are assigned executable code and a scheduling priority, and serve as the unit of isolation and execution. Communication between PDs is declared explicitly via communication channels (CCs), which support event notifications and protected procedure calls (PPCs). Shared memory communication is supported through explicitly declared memory objects.

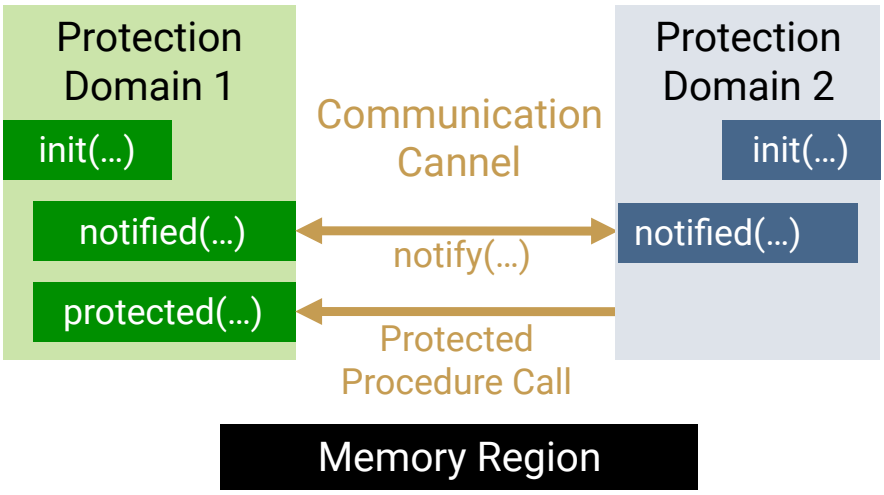

Figure 2.5: Microkit abstractions (Heiser, 2020)

2.3.3 LionsOS

LionsOS is a modular operating system designed for safety- and security-critical embedded systems, built on top of the seL4 microkernel and the Microkit framework. It aims to combine the formal assurance and isolation properties of seL4 with a practical, performant system design suitable for real-world deployment (Heiser et al., 2025).

Its design is guided by the principles of strict separation of concerns, least privilege, and *radical simplicity*, in line with the KISS ("Keep It Simple, Stupid") principle. Each

component is narrowly scoped and operates independently. Rather than supporting highly dynamic, general-purpose workloads, LionsOS targets statically architectured systems typical of embedded and cyber-physical domains, where resource allocation is known and determined at build time.

Unlike conventional OSes, LionsOS adopts use-case-specific policies instead of relying on one-size-fits-all designs (e.g., Linux). This approach avoids the complexity and inefficiencies typically associated with general-purpose policy frameworks.

In contrast to earlier attempts at modular, microkernel-based systems – which suffered from severe performance penalties due to high IPC and context-switching costs (Chen et al., 2024) – LionsOS shows that a simple and minimalist OS design can achieve competitive, or even superior, performance. Empirical evaluations have shown that LionsOS outperforms Linux in both throughput and latency on network-heavy workloads (Heiser et al., 2025).

2.3.4 The Cost of Verification

While LionsOS addresses many of the engineering and performance challenges traditionally associated with microkernel-based systems, the issue of formal verification remains a major barrier to achieving a provably secure operating system with a complete end-to-end verification story.

The experience of verifying seL4 demonstrated that functional correctness of real-world operating systems can be formally proven. However, the verification cost was substantial: verifying approximately 8,500 lines of C code required around 12 person-years of work, with an estimated cost of $350 per line of code (Klein et al., 2014). Although such an investment is justified for a operating system kernel, it would be impractical for verifying an entire operating system, which is significantly larger and much more complex.

Recent developments in automated verification have shown that significant reductions in verification effort are possible by leveraging SMT-based techniques instead of traditional interactive theorem proving (ITP). Approaches based on symbolic state-space exploration, guided by heuristics, allow many proof obligations to be discharged automatically without human intervention (Sigurbjarnarson et al., 2016; Zaostrovnykh et al., 2017; Nelson et al., 2017, 2019; Zaostrovnykh et al., 2019; Narayanan et al., 2020; Chen et al., 2023; Paturel et al., 2023; Cebeci et al., 2024). These techniques are particularly effective when applied to small, simple modules – an advantage made possible by LionsOS's minimal and modular system design.

However, even assuming that source-level verification becomes significantly more tractable through automation, a fundamental obstacle remains: ensuring that the compiled binary faithfully implements the verified source code. Without addressing the trustworthiness of the compiler, the overall system cannot achieve true end-to-end assurance.

In Section 2.4, we will discuss the implications of having the compiler as part of the trusted computing base, namely the challenges posed by unsafe language semantics, compiler optimisations, miscompilations, and the difficulty of ensuring that verified source-level properties are preserved at the binary level. We then examine approaches for addressing these issues through translation validation and verified compilation.

2.4 The Compiler Trust Problem

In his 1984 Turing Award Lecture, "Reflections on Trusting Trust," Ken Thompson revealed a disturbing possibility: even a fully open-source system can have undetectable backdoors if the compiler used to build it is compromised (Thompson, 1984). He described how a maliciously modified compiler could inject a backdoor into an operating system's `login` binary and, more insidiously, propagate this backdoor into future compiler binaries – even if the backdoor no longer appears in any source code. This "trusting trust" attack shows that no amount of source-level verification can guarantee system integrity if the compiler itself cannot be trusted.

Even putting aside malicious attacks, compilers are large and complex software systems, and like all such systems, they inevitably contain bugs. Despite decades of engineering effort, miscompilations – cases where a correct source program is silently translated into an incorrect binary – remain a recurring problem across all major compiler ecosystems. A large-scale study by Yang et al. (2011) found 325 previously unknown bugs across widely used C compilers, including GCC, Clang, and Intel CC, many leading directly to silent miscompilations.

Listing 2.1 shows a miscompilation introduced in GCC 9.1 during optimization. The program returns 0 without optimisations and 1 with them. The bug persisted until GCC 12.

```
int main(void) {
    for (unsigned a = 0, b = 0; a < 6; a += 1, b += 2)
        if (b < a)
            return 1;
    return 0;
}
```

Listing 2.1: Example of a miscompilation bug in GCC (Bug #102666)

For a provably secure system like **LionsOS**, this problem is not merely theoretical. A single miscompilation, whether due to a malicious injection, a compiler bug, or an unsound optimisation, can undermine the entire verification effort. Since proofs are typically constructed at the source level, the correctness of the resulting binary hinges entirely on the correctness of the compilation process itself. As such, we require techniques that remove the compiler from the TCB and therefore eliminate the need to trust it.

2.4.1 Producing a Verified Binary

As outlined by Leroy (Leroy, 2009), there are three widely used approaches for establishing compiler trustworthiness (i.e. removing the compiler from the TCB): *translation validation* (Pnueli et al., 1998), which can be implemented either via a *verified validator* or through *proof-carrying code* (Necula, 1997); and *compiler verification*.

Translation validation

In seL4's verification story, the compiler was removed from the TCB via a method known as translation validation, where for each individual compilation, the resulting binary is formally shown to refine the verified C source program (Sewell et al., 2013).

Rather than relying on a general formal semantics for C, which does not exist, seL4's verification fixes the semantics of the specific C program under verification using Norrish's StrictC parser (Tuch et al., 2007), producing a formal representation in Isabelle/HOL (Nipkow et al., 2002).

As shown in Figure 2.6, the compiled binary is then independently lifted into a corresponding HOL4 model using a verified disassembler built on a formal ISA specification (e.g. ARM, RISC-V). Both representations are translated into a common control-flow graph format, where semantic equivalence is established. Compiler optimisations are handled via rewrite rules in the prover.

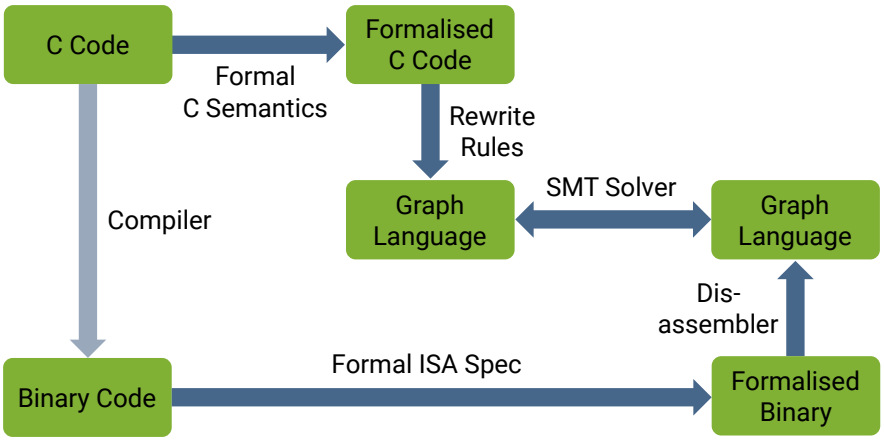

Figure 2.6: Translation validation proof chain (Heiser, 2020).

Equivalence between the two programs is then checked by splitting them into small chunks and discharging semantic preservation queries to SMT solvers. In theory, establishing full program equivalence is undecidable, as it reduces to the halting problem. In practice, however, the compiler's transformations are sufficiently regular to make the problem tractable for real-world code.

While effective, this approach is inherently fragile due to its dependence on the structure of the generated binary. The translation validation process relies on a number of assumptions about compiler behaviour that are not guaranteed to hold across versions or optimisation levels, making the method sensitive to changes in the compilation toolchain (Sewell et al., 2013).

This is an instance of the verified validator approach to translation validation, where a trusted validator proves each individual compilation correct. A closely related alternative is proof-carrying code (Necula, 1997), where the compiler itself produces a formal proof certifying the correctness of the compiled output. Conceptually, this shifts the role of validation from a standalone tool to the compiler: rather than verifying the binary after the fact, the compiler emits a proof – often in a general-purpose proof assistant such as Isabelle/HOL or Coq – that the output preserves the semantics of the input. This proof can then be checked independently, reducing the trusted base to the proof checker alone. In this setting, the compiler is treated as a certifying function that returns both a compiled binary and a proof object, and correctness requires only that the proof be valid with respect to the defined semantics. An example of this approach is the Cogent project (Amani et al., 2016), which we will briefly introduce in Section 3.2.

Verified compilation

A better alternative to translation validation is the use of a *verified compiler*, where it is guaranteed by construction that the generated binary preserves the semantics of the source program. Rather than verifying each individual compilation after the fact, the correctness of every compilation is a direct consequence of the compiler's proof.

Although compilers are large and intricate pieces of software, the property they must establish is surprisingly simple: the produced binary must refine the semantics of the input program. Formally, this can often be phrased as a simulation or refinement relation between the source and target languages, typically proved inductively on the structure of the compilation steps. In summary, every transformation applied by the compiler must be proven to preserve or refine the program's observable behaviour. Once this invariant is established for all stages, the compiler can be trusted for all programs within its specification (Leroy, 2009). This approach eliminates the fragility associated with translation validation and removes the compiler from the TCB entirely.

2.5 Verified Compilers

2.5.1 CompCert

CompCert is a widely-used, practical, production-grade verified optimising compiler for a substantial subset of the C programming language. Its correctness proof is mechanised in Rocq (previously known as Coq)(Bertot and Castéran, 2004) and formally

guarantees that the compiled assembly code behaves as specified by the semantics of the source C program. The verification covers the back-end optimisations and code generation phases. However, the front-end parsing and elaboration steps—which handle unsupported or ambiguous C constructs—are written in OCaml and are not verified. As a result, bugs found in the unverified front-end, as identified in (Yang et al., 2011), remain a residual risk.

Being a rather mature compiler, CompCert C supports most of ISO C99 and a significant portion of C11 (Leroy, 2009). Given its target domain of safety- and mission-critical systems, language coverage and compatibility have never been a major concern. In our experience, the engineering overhead associated with adopting CompCert for LionsOS is relatively low compared to alternatives.

Despite prioritising verification over aggressive optimisation, CompCert produces code that is highly competitive with traditional compilers. On PowerPC and ARM platforms, CompCert's generated code achieves about 90% of the performance of GCC 4 compiled at optimisation level `-O1`, and approximately 80–85% compared to `-O2` and `-O3`. The performance gap is primarily due to the absence of complex loop optimisations. Nonetheless, for most embedded and critical applications, this trade-off between performance and assurance is well justified (Leroy, 2009).

2.5.2 CakeML

CakeML (Kumar et al., 2014) is a verified functional programming language and toolchain based on a substantial subset of Standard ML (SML). It features a fully mechanised semantics in higher-order logic (HOL) and a verified compiler capable of bootstrapping itself (Tan et al., 2019). The language design is kept simple and practical to make formal verification tractable, although the standard basis library is still evolving.

The CakeML compiler consists of two frontends and a verified backend. The first frontend is a proof-producing translator that synthesises CakeML programs from HOL definitions, while the second is a conventional parser and type inferencer, both formally proved sound and complete. The backend generates machine code for several architectures, including x86, ARM, and RISC-V, and passes through multiple verified intermediate representations. The entire toolchain has been bootstrapped inside HOL, resulting in a verified binary that provably implements the compiler itself.

This bootstrapping approach means that, unlike CompCert, which relies on Coq's extraction mechanism to produce OCaml code and then invokes the unverified OCaml compiler to generate machine code, it implements a more comprehensive verification approach. CakeML is bootstrapped entirely "in the logic" – meaning the compiler's own implementation is processed through logical inference within the HOL theorem prover. This approach simultaneously produces the machine code implementation of the compiler and proves a theorem establishing its functional correctness, eliminating the need to trust any unverified code generation processes (Tan et al., 2019).

CakeML has been used to build a variety of verified applications, including Unix-like utilities (e.g., `grep`, `sort`, `diff`), proof checkers for certificates and SAT solvers (Tan et al., 2021), and larger systems like Candle, a verified re-implementation of the HOL Light theorem prover (Abrahamsson et al., 2022). It has also served as a basis for verified compilers targeting other languages, such as PureCake (for a lazy functional language) (Kanabar et al., 2023) and Pancake (for low-level systems programming) (Pohjola et al., 2023).

2.5.3 Pancake

Pancake is a recently developed verified systems programming language targeting low-level systems code, particularly device drivers (Pohjola et al., 2023). Designed to sit at an abstraction level between C and assembly, Pancake adopts a familiar C-like syntax as shown in Listing 2.2. Control flow constructs such as `while` loops, `if` statements, and function calls mirror their C counterparts, making the language accessible to systems programmers.

By design, Pancake prioritises ease of verification over language features, deliberately eschewing sophisticated type systems to simplify formal reasoning. The language design is radically simple. Pancake treats all data as machine words, with variable declarations using the `var N` syntax where `N` specifies the size in machine words (e.g., `var 1 dma_status` in Listing 2.2). There is no dynamic memory allocation at runtime, only a statically allocated memory region (referred to as the "heap" in Pancake terminology). There is no stack inspection, no concurrency primitives, and no undefined behaviour. Expressions are side-effect free, and evaluation order is fully specified, eliminating many sources of verification complexity common in C.

To interact with the outside world (e.g., Microkit and, by extension, seL4) or to implement features that are difficult to express natively or would perform poorly, Pancake uses foreign function interface (FFI) calls, denoted by the `@` prefix (e.g., `@assert(0,0,0,0)` in Listing 2.2).

```
fun handle_irq() {
    var 1 dma_status = get_dma_status();
    set_dma_status(dma_status);
    while (dma_status & DMA_INTR_MASK) {
        if (dma_status & DMA_INTR_RXF) {
            rx_return();
        }

        if (dma_status & DMA_INTR_TXF) {
            tx_return();
            tx_provide();
        }

        if (dma_status & DMA_INTR_ABNORMAL) {
            if (dma_status & DMA_INTR_FBE) {
                @assert(0,0,0,0); // FFI call
            }
        }

        dma_status = get_dma_status();
        set_dma_status(dma_status);
    }

    return 0;
}
```

Listing 2.2: Example of Pancake code.

Pancake has been used to verify a performant Ethernet device driver for the Lions operating system. The driver targets a 1GbE NIC on an ARM-based SoC, and achieves near line-rate performance with only around 10% CPU overhead compared to an unverified C implementation (Zhao et al., 2025).

Verification combines two mechanisms: (1) the aforementioned Pancake compilation pipeline, and (2) an automated deductive verification frontend based on Viper (Müller et al., 2016). Pancake source code is annotated with function contracts and loop invariants, which are preserved through transpilation into Viper's intermediate language for SMT-based verification. Memory safety, device interface compliance, network protocol correctness, and data integrity across packet transfers are all formally verified.

The implementation and verification for the entire ethernet driver took around three person-months of work by a systems engineer with no prior verification experience. The verification establishes end-to-end properties of the compiled binary, with the only major trusted components being the HOL4 theorem prover and the Viper verification framework.

This result demonstrates that Pancake is capable of supporting low-level, performant, verifiable device drivers without introducing excessive engineering or verification overhead.

Chapter 3

Related Work

In this chapter, we discuss other systems-level verification efforts and toolchains that also address the issue of trustworthy compilation.

3.1 CertiKOS

CertiKOS (Gu et al., 2016) is another major effort in operating system verification, targeting a relatively complex kernel with support for concurrency. Unlike seL4, which relies on translation validation to prove correctness of compiled binaries (Section 2.4.1), CertiKOS builds on a verified compilation pipeline (Section 2.5), using CompCertX[1], an extension of CompCert (Leroy, 2009) for concurrent low-level code.

Its success in achieving formal assurance down to the binary level demonstrates that verified compilation is a viable and effective approach. However, while the authors do provide evaluations of the kernel performance, they do not explicitly evaluate the impact of CompCert's overhead. That is, the performance cost introduced by using a verified compiler is not isolated, which is an understanding this thesis aims to provide.

3.2 Cogent

Cogent (O'Connor et al., 2016) is a purely functional, linearly typed language that enables certifying compilation by generating both C code and accompanying correctness proofs from high-level functional programs – an instance of the proof carrying code approach to translation validation (Necula, 1997). It was specifically designed to reduce the cost of verifying systems components such as file systems. To date, several file

[1] `https://github.com/CertiKOS/compcert`

systems have been implemented in Cogent for both Linux and seL4, including `ext2` (Amani et al., 2016).

The project shares many of the same goals as Pancake; however, it ultimately stalled, in part due to the restrictions imposed by its language design and the steep learning curve faced by systems engineers (Pohjola et al., 2023). To avoid these pitfalls, Pancake adopts a C-like semantics from the outset, making it more accessible. Nonetheless, it remains essential to carefully evaluate the engineering overhead involved to ensure the project remains practical for real-world systems development.

3.3 Verified OS Components in CakeML

Slind et al. (2020) presents a framework for building verified high-level OS components, such as input filters, using CakeML (Kumar et al., 2014) and deploying them on the seL4 microkernel (Klein et al., 2010). The approach allows developers to synthesise formally verified components – based on system-level specifications in AADL – and insert them around unverified legacy code to enforce safety and security properties. These components are compiled to verified binaries using the CakeML compiler and integrated into a seL4-based systems via the CAmkES framework (Kuz et al., 2007). This work demonstrates the applicability of CakeML in real-world systems verification; accordingly, our project explores similar ideas using CakeML in conjunction with Microkit, the successor to CAmkES.

3.4 The (Concurrent) Euclid Programming Language and the Toronto University System (TUNIS)

The Euclid and Concurrent Euclid programming languages were designed to support the construction of reliable and analysable systems software through strong typing, restricted aliasing, and disciplined modularity (Wortman et al., 1981; Holt, 1982a). TUNIS demonstrates how these language constraints can be applied in practice by re-implementing a Unix Version 7 compatible operating system with a clean, modular kernel architecture written entirely in Concurrent Euclid (Holt, 1982b). Although Euclid was created to facilitate formal verification, TUNIS itself was not fully formally verified. Instead it serves as a case study showing that a language designed for provability can support the development of realistic operating system components while improving clarity, structure, and reasoning about concurrency.

3.5 Why Rust Isn't the Answer (Here)

Rust is often religiously heralded as the solution to safe (systems) programming, with projects like Rust-for-Linux (Ojeda, 2021) and RedoxOS (Soller, 2015) reflecting its growing popularity in the OS community. Indeed, Rust provides a range of language-level safety guarantees – such as ownership-based memory management, lifetimes, and strict type checking – which help developers eliminate entire classes of memory and concurrency bugs. Given this, it is natural to ask: why not Rust? Why invest time and effort into building and using a new, immature programming language when Rust already promises safety, performance, and a growing ecosystem?

3.5.1 Safety $\neq$ Formal Verification

As introduced in Section 2.1, "complete formal verification is the only known way to guarantee that a system is free of programming errors" (Klein et al., 2010). While Rust offers strong safety guarantees at the language level, these do not amount to formal proofs of correctness. Provably secure systems require precise, machine-checked mathematical proofs about functional correctness – not just the absence of certain classes of bugs. In addition, low-level systems programming, such as device drivers, often requires the use of *unsafe* Rust, which reintroduces the very risks Rust aims to eliminate (Jung et al., 2018).

3.5.2 Verification Without End-to-End Guarantees

Recently, several automated verification frameworks have been developed to make system verification more practical. Tools like Prusti (Astrauskas et al., 2022) and Verus (Lattuada et al., 2024) allow developers to annotate Rust code with specifications and verify properties such as memory safety and functional correctness. These frameworks improve the tractability and scalability of verification, reducing the manual effort typically associated with interactive theorem proving. However, they operate on restricted subsets of Rust and rely on their own internal semantics, which are not formally linked to the behaviour of the Rust compiler. Since the Rust toolchain depends on `rustc` and LLVM, both of which are unverified and known to contain compiler bugs (Liu et al., 2025), there is no end-to-end guarantee that verified properties are preserved at the binary level. This is, fundamentally, the same problem that plagues verification efforts in C.

In fact, the same limitation applies to many earlier efforts in developing proof-oriented programming languages (e.g., SPARK Ada (Brunel et al., 2015), F* (Swamy et al., 2016)) and in retrofitting existing languages with automated verification frameworks (e.g., FramaC (2008)). Despite their verification capabilities at the source level, they all ultimately rely on an unverified code generation process. As Leroy (2009) argues,

verified compilation, and, translation validation, are the only known methods that can provide formal guarantees of semantic preservation from source to binary.

Chapter 4

Methodology

In this chapter, we provide an overview of the methodology used to evaluate the Pancake programming language's suitability for writing LionsOS components. Our evaluation will focus on both the engineering and performance overhead. As part of this analysis, we compare Pancake with traditional C compilers such as GCC and Clang, as well as with the verified CompCert C compiler, to understand how Pancake positions itself relative to these established toolchains

The core of our approach involves porting several significant LionsOS components. By re-implementing these components in Pancake and examining the resulting systems, we are able to assess development effort, code structure, and performance of a realistic system.

4.1 Target Components

We now introduce the target components used in our evaluation. We begin with an overview of these components and a brief description of their roles within LionsOS. We then discuss why these components are important for assessing Pancake's suitability, focusing on the aspects of functionality, complexity, and performance that make them representative and meaningful for our study.

4.1.1 The seL4 Device Driver Framework

One of the primary set of components we port in our evaluation is the seL4 Device Driver Framework (sDDF) (Heiser et al., 2024). It is a framework for building high-performance device drivers as user-level components on seL4 systems. It was created to address the long-standing reliability problems of traditional kernel-resident drivers, which form the majority of kernel code in monolithic systems (e.g. Linux) and are

a major source of operating-system vulnerabilities, accounting for the majority of the 1,057 CVEs reported for Linux in the period 2018–22 (Pohjola et al., 2023).

At its core, sDDF defines a set of interfaces, communication protocols, and design patterns that guide the structure of user-level device drivers. The framework centres around a lightweight shared-memory transport layer that connects drivers with client components such as network stacks or storage servers. Communication is organised through single-producer/single-consumer ring buffers, zero-copy data transfer, and semaphores for flow control. Drivers themselves are strictly single-threaded and focus solely on device-specific behaviour, while responsibilities such as device sharing, address translation, and cache management are handled by separate virtualiser components. This separation of concerns leads to small, modular components that compose into larger I/O paths. Figure 4.1 illustrates the typical structure of this architecture, showing how drivers, virtualisers, and client components interact through the transport layer.

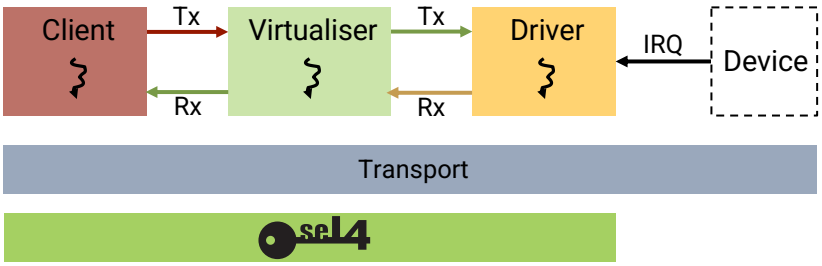

Figure 4.1: High-level structure of sDDF components and their communication model.

We selected sDDF for two main reasons:

1. Device drivers are among the most bug-prone components of an operating system, making them a natural priority in our verification efforts. Since sDDF provides a structured model for developing these drivers, it offers an appropriate setting to examine how well such components can be engineered in Pancake and to begin assembling a collection of Pancake-based, verification-ready drivers.
2. Device drivers are also some of the most performance-critical elements of an operating system. Evaluating an sDDF implementation in Pancake allows us to understand any performance overhead introduced by the language and to identify whether adopting Pancake for driver development would impact the practical efficiency of the I/O subsystem.

4.1.2 `libmicrokit`

As introduced in Section 2.3.2, Microkit is an SDK for constructing statically architectured systems on seL4. Each protection domain (PD) operates within a simple, uniform

event loop that processes notifications, messages, and faults according to the system's configuration. `libmicrokit` is the lightweight user-level library that implements this behaviour. It provides a thin layer of abstraction over the required seL4 system calls and exposes a uniform event-loop structure shown below.

```
for (;;) {
    seL4_Word badge;
    seL4_MessageInfo_t tag;

    if (have_reply) {
        tag = seL4_ReplyRecv(INPUT_CAP, reply_tag, &badge, REPLY_CAP);
    } else if (microkit_have_signal) {
        tag = seL4_NBSendRecv(microkit_signal_cap, microkit_signal_msg,
                              INPUT_CAP, &badge, REPLY_CAP);
        microkit_have_signal = seL4_False;
    } else {
        tag = seL4_Recv(INPUT_CAP, &badge, REPLY_CAP);
    }

    uint64_t is_endpoint = badge >> 63;
    uint64_t is_fault = (badge >> 62) & 1;

    have_reply = false;

    if (is_fault) {
        seL4_Bool reply_to_fault =
            fault(badge & PD_MASK, tag, &reply_tag);
        if (reply_to_fault) {
            have_reply = true;
        }
    } else if (is_endpoint) {
        have_reply = true;
        reply_tag = protected(badge & CHANNEL_MASK, tag);
    } else {
        unsigned int idx = 0;
        do {
            if (badge & 1) {
                notified(idx);
            }
            badge >>= 1;
            idx++;
        } while (badge != 0);
    }
}
```

Listing 4.1: Main `libmicrokit` event loop (`https://github.com/seL4/microkit`).

We include `libmicrokit` in our evaluation for two main reasons:

1. It plays an important role in our broader verification work. `libmicrokit` forms the interface layer through which user-level components, such as device drivers, interact with the seL4 kernel, making it a significant part of the trusted computing base.
2. It effectively serves as the runtime environment for all LionsOS components. Its performance influences the behaviour of the entire system, so porting its core logic provides insight into any potential performance implications of adopting Pancake for LionsOS.

4.2 Evaluation Approach

Having identified the components used in our study, we now outline the approach used to evaluate Pancake. Since our methodology is based on re-implementing existing LionsOS components, it is important to clarify what *porting* entails in the context of this thesis.

In this thesis, *porting* refers to re-implementing in Pancake the executable logic that defines the functional behaviour of each protection domain (PD). For Microkit-based components, this corresponds to the code reachable from the PD's two Microkit entry points, `notified` and `protected`. Conceptually, `notified` handles asynchronous notification events, while `protected` handles protected procedure calls (PPCs). These handlers implement the component's steady-state behaviour and represent the part of the system that executes continuously during normal operation. Our aim is to reproduce the structure and semantics of this behaviour as faithfully as possible in Pancake.

We do not port the device initialisation and other startup routines executed through the `init` hook of the C implementations. This choice is not due to engineering difficulty, since these routines typically consist of straightforward and repetitive configuration tasks. Rather, they provide little value for our study. They are not part of the main runtime execution path, they offer limited insight into Pancake's expressiveness, and they have no effect on the performance measurements we conduct. For these reasons, the porting effort focuses exclusively on the *functional core* of each component.

With this definition of porting in place, our evaluation proceeds along two complementary dimensions. The first is *engineering effort*, where we assess how naturally the selected components can be expressed in Pancake, what additional abstractions are required, and how the resulting implementations differ from their C counterparts. The second is *performance*, where we measure the runtime behaviour of Pancake implementations and compare them with C versions compiled using GCC, Clang and the verified CompCert compiler. This allows us to quantify any overhead introduced by using Pancake for systems code.

The remainder of this chapter elaborates on these two dimensions. We first describe the criteria and procedure used in the engineering evaluation. We then present the methodology for performance measurement, followed by a description of the experimental environment used in our analysis.

4.3 Engineering Evaluation

We now describe the methodology used to assess the engineering effort required to port LionsOS components to Pancake. Our evaluation combines quantitative metrics with qualitative observations.

4.3.1 Quantitative Metrics

We collect three primary metrics for each ported component:

1. **Lines of code**: We measure the source code size of both the original C implementation and the Pancake port. For Pancake, we separately count the initialisation code (which sets up the heap layout and global state) and the functional logic. This separation allows us to distinguish mechanical boilerplate from the core implementation effort. We report the relative difference as a percentage to quantify code expansion.
2. **Development time**: We record the effort required to complete each port in person days. This metric captures not only the complexity of the component but also the challenges encountered during development, such as debugging difficulties or unfamiliarity with the target protocol.
3. **Binary size**: We measure the size of the compiled ELF files for both C and Pancake implementations. This metric reveals the overhead introduced by the Pancake runtime, including FFI trampolines and other supporting infrastructure.

4.3.2 Qualitative Assessment

Beyond these metrics, we document our experience using Pancake throughout the porting process. We examine how naturally C idioms translate to Pancake, noting cases where the absence of language features complicates the implementation. We also assess toolchain maturity, including the quality of compiler error messages and the ease of debugging. These observations inform our recommendations for language and toolchain improvements that would reduce the barrier to adopting Pancake for systems development.

4.4 Performance Evaluation

We now outline the methodology used to assess the runtime performance of the Pancake implementations. Our approach benchmarks the Pancake ports within a small sDDF-based networking system and compares the results with the C and CompCert C versions.

4.4.1 Networking Example: Echo Server

For our performance evaluation, we use the networking example in LionsOS that builds an *echo server* using the sDDF networking subsystem. As shown in Figure 4.2, the setup exercises the full I/O path, comprising the Ethernet driver, RX and TX virtualisers, copiers, and an lwIP-based client that simply returns any received packet.

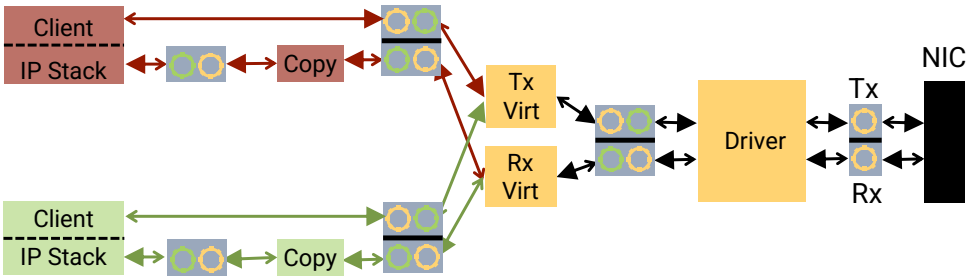

Figure 4.2: High-level structure of the networking example used in the performance evaluation.

4.4.2 Benchmarking Setup

We now describe the benchmarking setup used to evaluate the *echo server* networking example.

IPbench

To generate network load and record performance metrics, we use `ipbench` (Wienand and Macpherson, 2004), a benchmarking suite designed for reliable and repeatable evaluation of IP networks. It supports distributed load generation and does not require the target system to provide a full POSIX environment, which makes it well suited for benchmarking LionsOS components.

`ipbench` operates using three roles: a controller that coordinates each run, one or more client machines that generate the network traffic, and the target system that processes the traffic. The structure of this setup is shown in Figure 4.3.

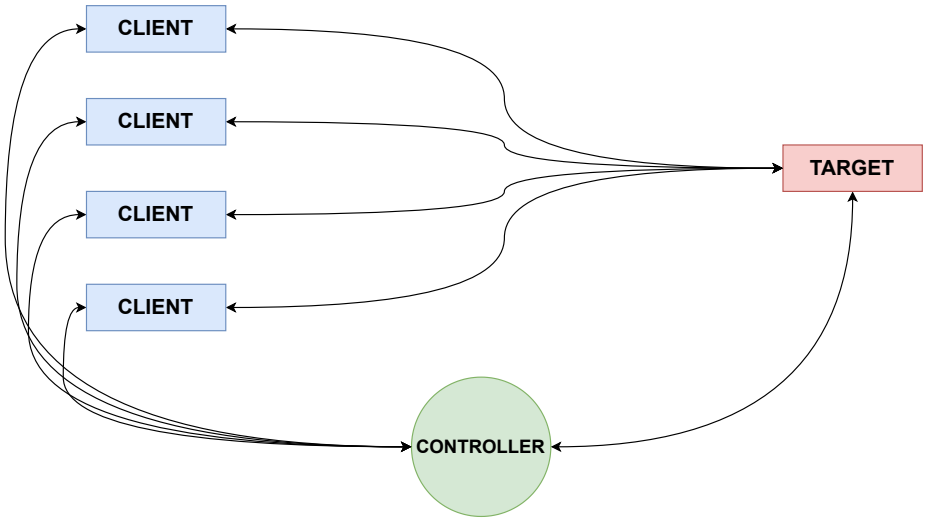

Figure 4.3: Structure of `ipbench`, showing the controller, load-generating clients, and the target system under test.

Collected metrics

We collect three groups of metrics:

- **Network metrics**: throughput at different offered loads and round-trip latency.
- **CPU utilisation**: overall and per–protection-domain utilisation, derived from PMU cycle counts together with idle-cycle measurements from the bench-idle PD.
- **Hardware counters**: raw performance monitoring unit (PMU) events such as total cycles, cache and translation lookaside buffer (TLB) misses, branch mispredictions, kernel entries, and cycles executed.

Experimental setup

We run all benchmarks on an Avnet MaaXBoard (Avnet Manufacturing Services, 2019) with an NXP i.MX8MQ SoC (NXP Semiconductors, 2021). The board has four Arm Cortex–A53 cores (up to 1.5 GHz), and we fix the clock at 1 GHz to avoid thermal throttling. It includes 2 GiB of RAM and an on-chip 1 Gb/s Ethernet controller, which we use for all networking experiments. All tests use UDP traffic.

We build the *echo server* system using Microkit `2.0.1` in the non–SMP configuration and with the `benchmark` build mode enabled. We compile the C baseline with Clang `21.1.4` and compile the CompCert C baseline with CompCert `3.16`. Pancake

build uses the CakeML compiler built from `b76a67d`, together with HOL4 at `72e78b2` and Poly/ML `5.9`.

The `ipbench` load generators run on a Linux host (kernel `<insert version>`), which also acts as the controller that coordinates each benchmark run. We measure throughput at offered loads of 10 Mb/s, 20 Mb/s, 50 Mb/s, 100 Mb/s, and increasing levels up to 970 Mb/s. We stop at 970 Mb/s because with the default payload size of 1470 B and roughly 40 B of Ethernet+IP+UDP header overhead per packet, the achievable line rate is capped at $1470/(1470 + 40) \simeq 0.9735$. Before each experiment, `ipbench` performs its standard warm-up phase. We repeat the entire experiment (covering all throughput levels) 10 times and report the mean and standard deviation across these repetitions.

Reproducibility All code used in our benchmarks, including the Pancake ports and the C and CompCert baselines, is available at `https://github.com/zhewenshen/sddf`. The benchmarking scripts, including the driver for running `ipbench` experiments, are included in the same repository. One can reproduce the results by running `autobench.py` with the required arguments, assuming access to the benchmarking infrastructure.

Chapter 5

Implementation Experience

This chapter describes our work in porting OS components to Pancake. We outline our approach and the steps we follow during the port. We then discuss the difficulties and challenges we face and give feedback on possible improvements to the language and its workflow. The chapter ends with an overview of possible automations that could reduce manual effort and improve the scalability and future adoption of the Pancake programming language.

We will not make comments or comparisons on our experience with CompCert, since the workflow mostly mirrors regular C and offers little additional insight.

5.1 How to do a Pancake port?

We will now provide an overview of the typical procedure we follow when we port a LionsOS component to Pancake. The process begins with an analysis of all global variables that must be initialised on the C side. These often include base register addresses, queue pointers, queue sizes, channel IDs, and any other component specific structures. We then map these values onto the Pancake heap, which is just the static region in the final program. After this setup we begin the main porting process. We work in an iterative manner. We start with the outermost function, which serves as the entry point to the protection domain. For drivers this is usually the notified entry point. We port this function to Pancake and leave all functions that it calls as FFI calls into the original C code. After testing this stage we select one of these FFI functions and port it to Pancake, while leaving any inner calls as FFI for the moment. We repeat this process until the main logic of the component has been fully ported. Once the core functionality is tested and confirmed to work, we carry out a final clean up step and move any constant global values into Pancake globals.

5.2 Development Experiences with Pancake

We now describe our experience porting LionsOS components to Pancake. Over the course of this project, we ported the majority of sDDF components, including device drivers for Ethernet, serial, timer, and I^2C, block across multiple hardware platforms, as well as the virtualiser components for both the networking and serial subsystems. Table 5.1 summarises the scope of our porting effort along with key metrics for each component.

Driver	C LoC	PNK Init	PNK LoC	PNK Total	Rel. Diff. (%)	P-Days
Ethernet Drivers						
IMX	227	48	262	310	+36.56	1
MESON	227	52	279	331	+45.81	3
DWMAC	273	180	259	439	+60.81	7
Serial Drivers						
ARM	120	30	169	199	+65.83	2
IMX	133	39	166	205	+54.14	1
MESON	142	39	154	193	+35.92	1
NS16550A	114	32	173	205	+79.82	1
ZynqMP	190	35	170	205	+7.89	1
Timer Drivers						
ARM	128	75	152	227	+77.34	2
Goldfish	84	31	113	144	+71.43	1
IMX	100	39	163	202	+102.00	1
MESON	96	45	104	149	+55.21	1
JH7110	133	35	154	189	+42.11	1
I2C Drivers						
MESON	333	158	417	575	+72.67	5
Network Components						
Virt RX	140	75	211	286	+104.29	2
Virt TX	101	55	143	198	+96.04	2
Copy	71	45	91	136	+91.55	1
Serial Components						
Virt RX	86	60	119	179	+108.14	2
Virt TX	114	64	240	304	+166.67	2
Summary	2,593	1,163	3,139	4,302	+65.91	37

Table 5.1: Comparison of lines of code between C and Pancake implementations, with Pancake development effort in person days. Components such as virtio and block drivers are omitted as their debugging remains ongoing.

5.2.1 Development Effort Analysis

Table 5.1 summarises our porting effort across all components. The average code expansion of 65.91% reflects several factors inherent to Pancake: explicit memory management, the absence of high level abstractions such as structs, and the initialisation boilerplate required to set up the heap layout. The initialisation code (PNK Init column) is largely mechanical and we can adapt it from existing ports with minimal modification.

Development time varied significantly across components. While most drivers required only one to two person days after initial familiarity with the language, we observed a small number of higher-effort ports. The learning curve for Pancake is noticeable but manageable. Once we internalised the idioms and memory layout conventions, our productivity improved considerably. However, the majority of excess effort in our experience stemmed not from the language itself, but from difficulties in debugging.

5.2.2 Compiler Error Messages

A significant obstacle throughout development was the quality of compiler error messages. In practice, we encountered only two categories of error output:

```
### ERROR: parse error
Parse tree conversion failed at unknown location
Program exited with nonzero exit code
```

Listing 5.1: Pancake parse error without location information

```
### ERROR: parse error
Not combinator failed at line 18
j = j - 1;
^
Program exited with nonzero exit code
```

Listing 5.2: Pancake parse error with line information

These messages provide minimal diagnostic information. The underlying cause could range from a missing semicolon, an omitted return statement, or the use of unsupported syntax, yet all produce nearly identical output as shown in Listing 5.1 and Listing 5.2. More problematically, the reported line numbers are frequently inaccurate, particularly in the presence of deeply nested loops or conditional blocks. We often resorted to manually bisecting the source file to isolate the offending line, a tedious process that consumed considerable development time.

5.2.3 Debugging at Runtime

For more complex components, compile time errors were only part of the challenge. Understanding runtime behaviour, particularly for stateful drivers, required extensive use of printf-style debugging. However, Pancake lacks native support for string literals or formatted output, so we must perform all debugging output via FFI calls to C.

Our workaround was to implement a debug FFI function that prints an integer "debug ID" together with a value to inspect. Since Pancake does not support string literals, the debug ID acts as a stand-in for a descriptive message (e.g., "XXX returned value of %d"). A typical debugging session thus involves inserting calls such as those shown in Listing 5.3.

```
@debug_print(0, 1, 0, state);
@debug_print(0, 2, 0, counter);
```

Listing 5.3: Debug printing in Pancake using numeric debug IDs (in place of strings)

We must then map each numeric debug ID back to the corresponding source location (or intended message), a process that is error prone and scales poorly as the number of debug points increases. The absence of even basic string support makes printf-style debugging, a staple of systems development, remarkably cumbersome.

5.2.4 Case Study: I^2C Driver

The I^2C driver required five person days, considerably more than comparably sized components. Three factors contributed to this overhead. First, the underlying C implementation is inherently more complex than the network or serial drivers, involving a state machine to manage multi phase transactions. Second, representing this state machine in Pancake without named structures or enumerations proved error prone; state variables become anonymous offsets into the heap, and any indexing mistake silently corrupts unrelated data. Third, our unfamiliarity with the I^2C protocol itself compounded these difficulties, as distinguishing driver bugs from protocol misunderstandings required significant investigation.

5.2.5 Case Study: DWMAC Ethernet Driver

The DWMAC ethernet driver represents our most challenging port, requiring seven person days. The difficulty did not stem from the driver's complexity, but from a compiler bug in Pancake's code generator.

Pancake has an experimental unverified feature called multiple entry points, which is crucial to LionsOS components. However, there is a bug with RISC-V code generation

where it makes a false assumption about register usage across entry points and fails to adhere to the RISC-V calling convention. Specifically, the generated code does not preserve and restore callee saved registers correctly, leading to silent corruption when the runtime invokes different entry points in sequence.

This bug was particularly insidious because the symptoms manifested as driver misbehaviour, leading us to initially suspect errors in our own code. We spent considerable time reviewing the Pancake source and comparing against the working C implementation before eventually inspecting the generated assembly. We only identified the root cause after observing anomalous register values that our source level logic could not explain.

We developed an ad hoc workaround that involved hex editing the compiled binary to replace problematic register references with temporary registers that did not require preservation. This brittle solution underscores the challenges of working with an experimental compiler. We have since reported the bug to the Pancake developers and they are currently addressing it.

5.2.6 Binary Size Overhead

Beyond source code expansion, we also measured the impact of Pancake on compiled binary size. Table 5.2 presents the ELF file sizes for both C and Pancake implementations across all ported components.

Driver	C ELF	PNK ELF	Overhead (%)
Ethernet Drivers			
IMX	21,238	45,350	+113.5
MESON	21,600	45,718	+111.7
DWMAC	25,969	52,560	+102.4
Serial Drivers			
IMX	20,490	62,831	+206.6
MESON	21,386	63,199	+195.5
NS16550A	15,485	57,243	+269.7
Timer Drivers			
IMX	20,845	54,353	+160.7
MESON	20,639	54,315	+163.2
JH7110	15,736	52,723	+235.0
I2C Drivers			
MESON	22,754	55,382	+143.4
Network Components			
Virt RX	23,856	71,381	+199.2
Virt TX	23,584	63,057	+167.4
Copy	17,520	63,139	+260.4
Serial Components			
Virt RX	23,684	60,798	+156.7
Virt TX	34,843	73,062	+109.7
TOTAL	329,629	875,111	+165.5

Table 5.2: ELF size comparison (in bytes) between Pancake and C implementations.

The average overhead of 165.5% is substantial, but the increased binary size does not directly correspond to the increased lines of code. Several factors contribute to the bloated ELF files. First, the initialisation code that sets up the heap layout, configures base addresses, and prepares data structures constitutes dead code during normal runtime execution. This setup runs once at startup and the code remains resident in memory thereafter.

Second, Pancake inherits infrastructure from the CakeML ecosystem, including a garbage collector that our components never utilise. LionsOS drivers operate with static memory allocation and do not require garbage collection, yet the collector code is linked into every Pancake binary regardless.

Third, each FFI call requires a trampoline to transition between the Pancake runtime and native C code. These trampolines add overhead proportional to the number of distinct FFI functions we invoke. Likewise, the actual C implementations of our FFI functions are link

5.2.7 Engineering Overhead

This section examines the main language limitations we encountered when porting LionsOS components to Pancake. Whilst Pancake can express all required functionality, several areas introduce notable development burden. The primary challenges are: the representation of structured data, the absence of division and modulo operators, constraints in the foreign function interface, and the manual coordination of static memory layouts. These limitations do not prevent implementation, but they require workarounds that increase manual effort and demand explicit reasoning about correctness. Errors in memory layout coordination are particularly difficult to debug, often manifesting only at runtime. We describe each challenge and discuss its practical impact on the porting workflow outlined in the previous section.

Named structures In C, a `struct` defines a named collection of related fields that share a contiguous block of memory. This gives each field a clear role in the program and makes the structure easy to understand and maintain. Pancake originally provided only an array like structure, where each field had to be accessed through its numeric index. An example is shown in Listing 5.4.

```
var 3 myStruct = <1, 2, 3>;

var firstItem = myStruct.0;
var secondItem = myStruct.1;
var thirdItem = myStruct.2;
```

Listing 5.4: Accessing elements of a Pancake array based structure

This representation behaves more like an untyped array than a structured data type. Since all fields are identified only by their position, the connection between the stored values and their meaning is lost. This reduces readability and makes mistakes easier to introduce. In our I^2C driver, the original C state structure shown in Listing 5.5 could not be expressed in a clear way using the array based form.

```
typedef struct _i2c_ifState {
    uint8_t *curr_data;
    int curr_request_len;
    int curr_response_len;
    size_t remaining;
    bool notified;
    uint8_t rw_remaining;
    enum data_direction data_direction;
    size_t addr;
} i2c_ifState_t;
```

Listing 5.5: The I^2C interface state structure in C

To retain clarity, we instead flattened the structure into a set of global variables, each with a descriptive name, as shown in Listing 5.6. This avoids the confusion caused by positional indices and keeps the intent of the data explicit.

```
var 1 g_curr_data = 0;
var 1 g_curr_request_len = 0;
var 1 g_curr_response_len = 0;
var 1 g_remaining = 0;
var 1 g_notified = 0;
var 1 g_rw_remaining = 0;
var 1 g_data_direction = 0;
var 1 g_addr = 0;
```

Listing 5.6: Flattened Pancake globals used to replace the C structure

This approach improves readability, but it introduces extra manual bookkeeping and becomes harder to maintain as the number of fields grows. Macros could map names to array indices, but this approach is fragile and can silently break if the layout changes. Since then, the Pancake team has added support for named structures, which aligns the language more closely with C and avoids the need for these workarounds in future development.

Division and modulo Pancake does not support division or modulo operations. The language provides only addition, subtraction, multiplication, and bitwise operations. This limitation primarily affects timer-related code in LionsOS, where conversions between hardware tick counts and time values require division. These conversions involve two cases: converting ticks to nanoseconds and converting nanoseconds back to ticks. The mathematical structure of these conversions determines which approximation strategies work in Pancake. The trivial workaround is to use an FFI call, however given one of the aim of this thesis is to evaluate the expressiveness of Pancake, we instead explore what can be achieved natively.

Converting ticks to nanoseconds uses the formula `ticks * NS_IN_US / GPT_FREQ`, where `NS_IN_US = 1000` and `GPT_FREQ = 12`. This simplifies to `ticks * 1000 / 12`. The ratio 1000/12 is greater than 1, truncating to 83 in integer division. Because the ratio exceeds 1, we can approximate this entire expression in Pancake with a single integer multiplication, as shown in Listing 5.7.

```
var ticks = get_ticks();
var time_ns = ticks * 83;  // 1000 / 12 truncates to 83
```

Listing 5.7: Explicit approximation when converting ticks to nanoseconds

The inverse conversion from nanoseconds to ticks uses `timeout_ns * 12 / 1000`. Here, the ratio $12/1000 = 0.012$ is less than 1, which fundamentally changes the problem. We cannot represent a value less than 1 as an integer multiplier. The solution is to find an equivalent fraction with a power-of-two denominator. Since $12/1000 \sim 3/256$, we can express this as a multiplication by 3 followed by a right shift by 8 bits, as shown in Listing 5.8.

```
var offset_ticks = (timeout_ns * 3) >> 8;  // 12 / 1000 ~ 3 / 256
var new_timeout = curr_time + offset_ticks;
```

Listing 5.8: Power-of-two fraction approximation when computing timeout values

Modulo operations face similar constraints. Pancake provides no built-in modulo operator, so we must use alternative techniques. When the divisor is a power of two, a bitwise mask produces the modulo result directly. Listing 5.9 shows this pattern for a circular queue index where `rx_hw_cap` must be a power of two.

```
var idx = rx_tail & (rx_hw_cap - 1);
```

Listing 5.9: Modulo by a power of two using a bitwise mask

For divisors that are not powers of two, no efficient pattern exists. We must implement repeated subtraction in a while loop, as shown in Listing 5.10. This approach is verbose and runs extremely slowly, particularly when the dividend far exceeds the divisor.

```
var result = value;
while (result >= 10) {
    result = result - 10;
}
// result now contains value % 10
```

Listing 5.10: Computing modulo for non-power-of-two divisors using a while loop

These workarounds suffice for all current LionsOS components. However, the engineering overhead is high. We must now reason explicitly about the correctness and validity of simple operations like division and modulo that would otherwise be straightforward in C.

Foreign function interface Pancake provides a foreign function interface to call C functions for operations that cannot be easily implemented in Pancake itself. The

primary use case is system calls into the seL4 kernel and the Microkit abstractions built on top of them. These core FFI calls are relatively straight forward and present no significant issues. However, FFI limitations become problematic during porting and debugging, when we temporarily need to FFI out code sections that have not yet been fully translated to Pancake. These ad-hoc FFI calls expose two significant constraints: a restriction to four arguments and the absence of return values.

The four-argument restriction arises from Pancake's FFI design, which allows at most four parameters per call. When temporarily FFI-ing out code during porting or debugging, we often encounter sections that would require more than four arguments. To work around this, we need to store additional values in Pancake's static memory region before invoking the FFI function, then access them from the C side through the shared static memory array. This manual parameter passing is verbose and requires careful coordination of memory layout between Pancake and C code.

The absence of return values presents a similar challenge. All FFI functions follow the signature `void ffifn(unsigned char *c, long clen, unsigned char *a, long alen)`, with no mechanism for returning values directly. When we need a return value, we must implement a manual return protocol. The C function writes the result into a predetermined location in the shared static memory region, and the Pancake code subsequently loads it from that location. This pattern effectively implements explicit return value passing through shared static memory.

Consider the example from the libmicrokit event loop port, where we need to check whether a signal has occurred. The Microkit runtime maintains a global variable `microkit_have_signal` that indicates signal status. To retrieve this value in Pancake, we define an FFI function that stores it at a specified address:

```
void ffimicrokit_have_signal(unsigned char *c, long clen,
                             unsigned char *a, long alen) {
    pnk_mem[clen] = microkit_have_signal;
}
```

Listing 5.11: FFI function storing a return value in shared static memory

Here, `pnk_mem` is the shared static memory region visible to both C and Pancake, and we use the `clen` parameter to specify where to store the value. On the Pancake side, we invoke this FFI function with `HAVE_SIGNAL_ADDR` as the storage location, then load the result:

```
@microkit_have_signal(0, HAVE_SIGNAL_ADDR, 0, 0);
var microkit_have_signal = lds 1 @base + HAVE_SIGNAL_ADDR * @biw;
```

Listing 5.12: Invoking FFI and loading the return value from shared static memory

This approach requires careful coordination of addresses and memory layout between C and Pancake code. Any mismatch in the assumed memory layout introduces potential addressing errors that may only manifest at runtime.

Static memory layout coordination The memory coordination challenges described in the FFI section extend to the broader setup of Pancake's static memory region. During porting, we must manually establish a mapping between C global variables and their corresponding locations in Pancake's static memory. This mapping defines where each global structure, array, or value resides and how much space it occupies. Any error in this setup creates immediate problems. If we assign variables to overlapping memory regions or fail to keep the C and Pancake layouts synchronised after modifications, the component will fail at runtime. These failures manifest as incorrect values, memory corruption, or unexpected behaviour that provides little indication of the underlying cause. The bug is not intuitive to diagnose because the code itself may be correct whilst the memory layout assumptions are wrong. Debugging requires carefully tracing through both the C initialisation code and the Pancake memory access patterns to identify where the layouts diverge. This coordination overhead is particularly challenging during iterative porting, where we frequently modify which portions of code run in C versus Pancake, requiring corresponding updates to the memory layout on both sides.

5.3 Towards Automation: `C2Pancake` Transpiler

Much of the engineering overhead described above stems from the manual nature of the porting process. The cognitive load of tracking memory layouts, managing FFI coordination, and reasoning about language limitations creates opportunities for human error. A natural solution is to automate as much of this process as possible, reducing both the effort required and the errors introduced by manual translation. Whilst not originally a goal of this thesis, we developed a prototype implementation of such a tool, which we refer to as the C2Pancake transpiler.

The approach follows a standard transpilation workflow inspired by projects such as `C2Rust` (Immunant, Inc., 2024). We extract the abstract syntax tree from C source code using `libclang`, then walk this AST to generate corresponding Pancake code. The flow is illustrated in Figure 5.1.

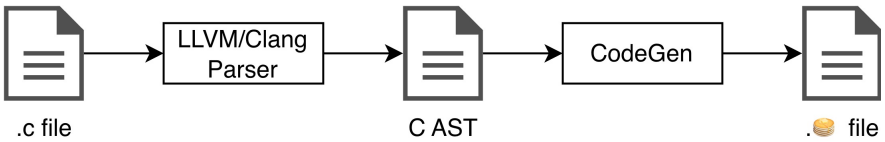

Figure 5.1: Prototype `C2Pancake` transpiler workflow

The current prototype supports basic function translation, including variable assign-

ments, arithmetic operations, and loop conversions. Since Pancake supports only while loops, we transform both for loops and while loops from C into Pancake's while loop construct. However, this prototype makes a significant simplifying assumption: it treats all variables as local to Pancake. This means we avoid the memory layout management issues discussed earlier, as the tool does not handle global variables or the coordination between C and Pancake static memory regions.

Listing 5.13 shows an example translation from C to Pancake. The transpiler converts the C for loop into a while loop, explicitly managing the loop variable initialisation and increment. The if-else structure is preserved, and compound assignment operators are expanded into their explicit forms.

C Source:

```c
// For loop with if-else
for (int j = 10; j > 0; j--) {
    if (j > 5) {
        int y = j - 1;
    } else {
        int z = j + 1;
    }
}
```

Generated Pancake:

```
var j = 10;
while (j > 0) {
    if (j > 5) {
        var y = j - 1;
    } else {
        var z = j + 1;
    };
    j = j - 1;
}
```

Listing 5.13: Example C to Pancake translation showing for-loop conversion

Extending this prototype to handle real-world porting would require addressing several challenges. First, we need a mechanism for managing static memory layouts. The most practical approach is to require developers to annotate global variables with their intended memory locations, allowing the transpiler to generate the correct access patterns whilst keeping the human in control of the overall layout. Second, we need to determine which functions should remain as FFI calls and which should be transpiled. Again, annotations provide a straightforward solution, letting developers mark functions that must stay in C. Third, we must handle unsupported language features. C code frequently uses constructs that have no direct equivalent in Pancake, such as for loops with complex initialisation and increment expressions, or switch statements. One could develop transformation rules and macro systems to convert these constructs automatically, but this approach is fragile and must handle numerous edge cases. A more practical solution is to require developers to manually rewrite the source C code to eliminate unsupported features before running the transpiler. This keeps the transpiler simple whilst still providing substantial automation benefits for the supported subset of C.

5.3.1 Vision for a Complete Transpiler

An ideal `C2Pancake` transpiler would significantly reduce the porting effort described in this chapter. Such a tool would accept annotated C source code and produce verification-ready Pancake code with minimal manual intervention. Developers would annotate global variables with memory layout information and mark FFI boundaries. The transpiler would then handle the mechanical work: generating memory access patterns, transforming control flow, managing calling conventions, and validating memory layouts. It would warn about potential issues such as overlapping regions or unsupported features requiring manual refactoring. This would liberate developers from tedious mechanical translation and debugging, making the approach far more scalable for larger codebases.

We leave the detailed design of such a tool as future work, as the Pancake language continues to mature and new features are added that may simplify or alter the transpilation strategy.

Chapter 6

Performance Evaluation

This chapter presents results from the experiments described in Section 4.4.2. We analyse the performance overhead of Pancake compared to C and CompCert baselines, and explore optimisation techniques to reduce this overhead.

6.1 sDDF Networking Performance

6.1.1 Baseline Measurements

This section compares the unoptimised implementations. The C baseline is compiled with Clang, and the same C code compiled with CompCert establishes the expected overhead of verified compilation. The Pancake implementation is a direct translation from the C source with no additional optimisations applied. Measurements span two ranges: low throughput (10–100 Mb/s) and full throughput (10–970 Mb/s). At higher loads, batching amortises per-packet costs and can mask performance differences, so the low throughput range better isolates per-packet overhead.

Figure 6.1 shows throughput and CPU utilisation across the load range. All three implementations achieve identical throughput, confirming that Pancake introduces no functional bottleneck. The differences appear in CPU utilisation: as summarised in Table 6.1, CompCert adds +2.6% overhead at low throughput and +1.4% at full throughput, whilst Pancake shows considerably higher overhead at +8.2% and +4.2% respectively.

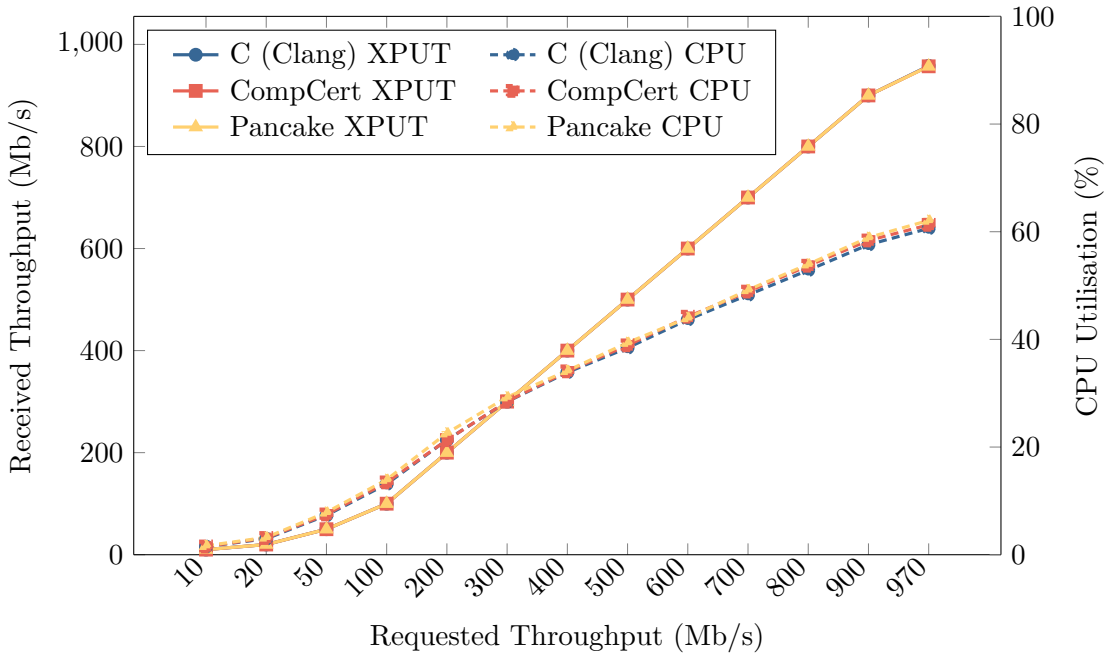

Figure 6.1: Throughput and CPU utilisation of the echo server at varying requested throughput

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+2.6%	0.0%	+1.4%
Pancake	0.0%	+8.2%	0.0%	+4.2%

Table 6.1: Relative difference in throughput and CPU utilisation compared to C (Clang) baseline

Figure 6.2 confirms that this CPU overhead does not translate into user-visible latency degradation—all implementations exhibit comparable round-trip times at equivalent CPU utilisation levels. Pancake's slight RTT elevation at 500 Mb/s corresponds to its higher CPU utilisation, which matches the level C reaches at 600 Mb/s.

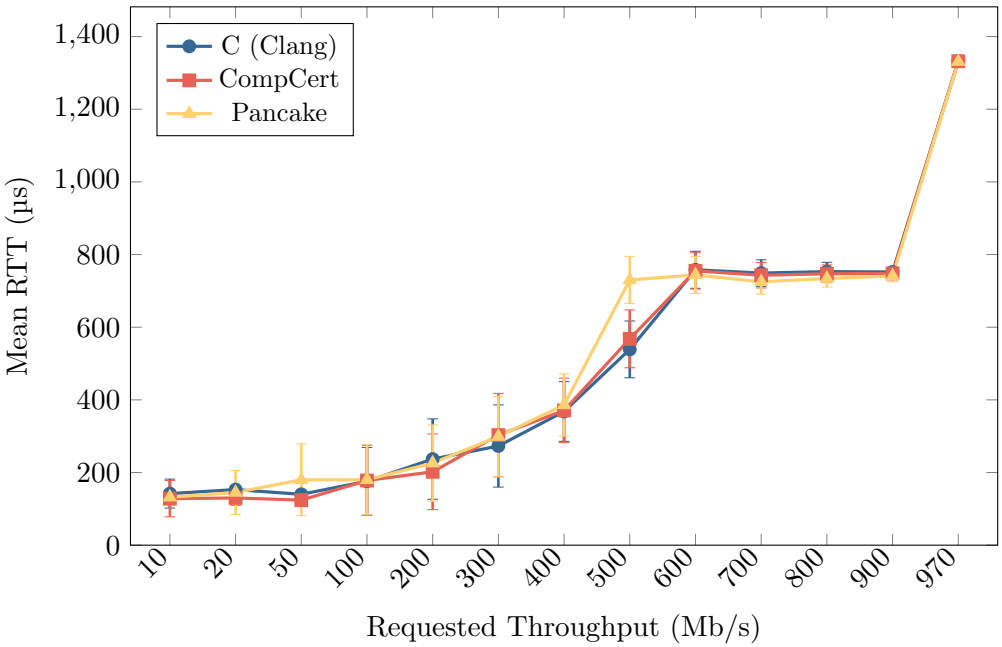

Figure 6.2: Mean round-trip time of the echo server at varying requested throughput

To quantify the driver-level overhead, Figure 6.3 isolates the Ethernet driver. As shown in Table 6.2, Pancake adds +25.0% overhead at low throughput and +14.1% at full throughput, compared to just +4.9% and +2.6% for CompCert. The subsequent sections investigate the sources of this overhead and evaluate optimisations to address them.

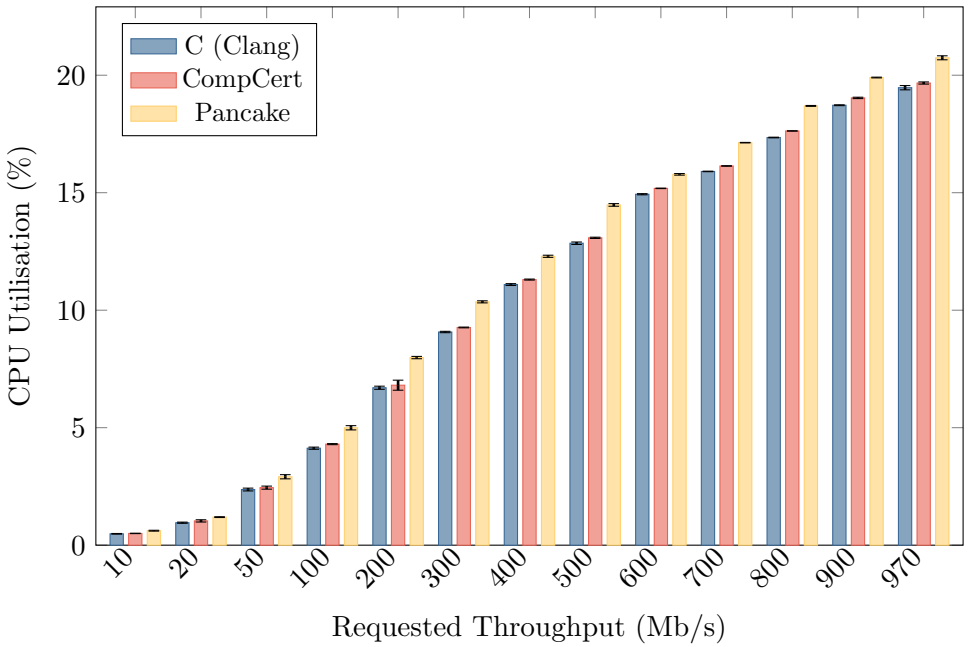

Figure 6.3: CPU utilisation of the Ethernet driver at varying requested throughput

	Low Throughput (10–100 Mb/s)	**Full Throughput (10–970 Mb/s)**
CompCert	+4.9% (+0.06 pp)	+2.6% (+0.18 pp)
Pancake	+25.0% (+0.30 pp)	+14.1% (+1.00 pp)

Table 6.2: Relative CPU utilisation overhead of the Ethernet driver compared to C (Clang) baseline

6.1.2 Inline Optimisation

The driver code contains many small helper functions that C compilers typically inline. For example:

```
static inline bool hw_ring_full(hw_ring_t *ring)
{
    return ring->tail - ring->head == ring->capacity;
}
```

Pancake includes an inline feature, but it has not yet merged upstream and remains under testing by the Pancake team. To evaluate the potential impact of inlining without depending on this experimental feature, we use C preprocessor macros to manually expand function calls to their bodies directly in the Pancake source.

Figure 6.4 and Table 6.3 show the results with manual inlining applied. In terms of *relative CPU-utilisation overhead* (vs the C/Clang baseline), Pancake improves from +25.0% to +20.1% at low throughput, and from +14.1% to +11.3% at full throughput. This represents a noticeable improvement, but Pancake still has roughly 4 × the *overhead* of CompCert at low throughput (20.1% vs 4.9%) and about 4.3 × at full throughput (11.3% vs 2.6%). We conclude that function call overhead contributes to the performance gap but is not the primary cause.

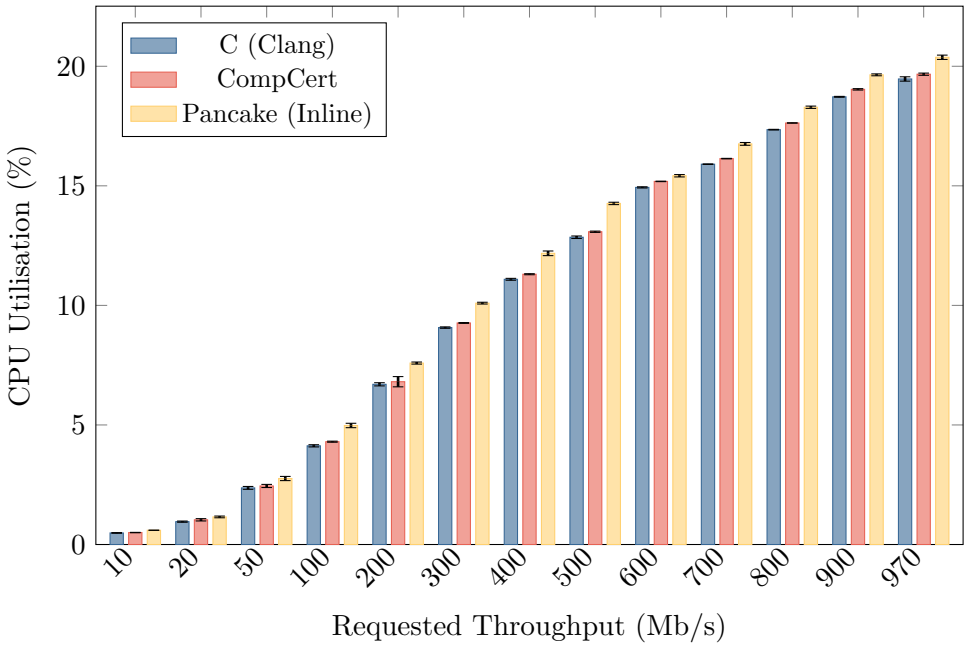

Figure 6.4: CPU utilisation of the Ethernet driver with inline optimisation

	Low Throughput (10–100 Mb/s)	**Full Throughput (10–970 Mb/s)**
CompCert	+4.9% (+0.06 pp)	+2.6% (+0.18 pp)
Pancake (Inline)	+20.1% (+0.23 pp)	+11.3% (+0.77 pp)

Table 6.3: Relative CPU utilisation overhead of the Ethernet driver with inline optimisation compared to C (Clang) baseline

6.1.3 Global Variable Caching

During packet processing, the driver frequently accesses global variables such as ring buffer pointers, queue capacities, and device register addresses. Many of these values remain constant or invariant within a function invocation. Modern C compilers optimise such accesses by caching frequently used globals in registers, avoiding repeated memory loads. Pancake's global variable support is relatively new and does not yet perform this optimisation, so each access to a global variable results in a load from memory.

We identified many instances in the driver code where globals are accessed repeatedly within loops, roughly following this pattern:

```
// Before: repeated global loads
while (true) {
    x = load(some_global) + 1;
    y = load(another_global) * 2;
}

// After: cached globals
cached_a = load(some_global);
cached_b = load(another_global);
while (true) {
    x = cached_a + 1;
    y = cached_b * 2;
}
```

To evaluate the impact of this overhead, we manually cache global variables by creating local shadow copies at the start of each function, then use these locals throughout the function body. This transformation reduces the number of memory loads from $O(n \times g)$ to $O(g)$ per function call, where n is the number of loop iterations and g is the number of globals accessed.

Figure 6.5 and Table 6.4 show the driver-level results with both inlining and global caching applied. The improvement is substantial: Pancake's overhead drops from +20.1% to +6.5% at low throughput, and from +11.3% to +4.6% at full throughput. This brings Pancake within range of CompCert, which shows +4.9% and +2.6% overhead at low and full throughput respectively. We conclude that repeated global variable loads, rather than function call overhead, were the primary source of Pancake's baseline performance gap in the context of LionsOS ethernet drivers.

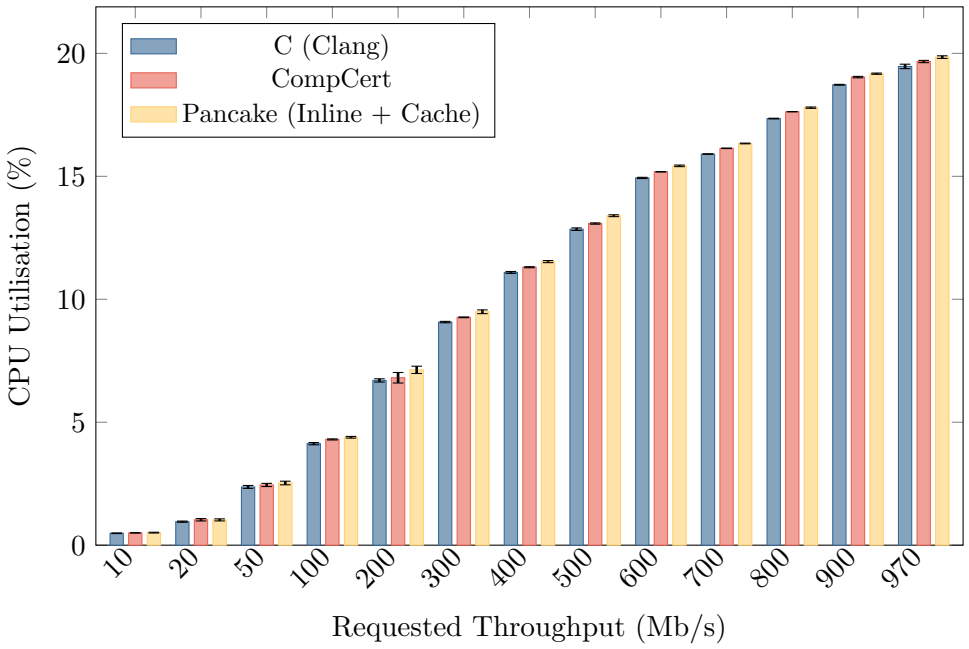

Figure 6.5: CPU utilisation of the Ethernet driver with inline and global caching optimisations

	Low Throughput (10–100 Mb/s)	**Full Throughput (10–970 Mb/s)**
CompCert	+4.9% (+0.06 pp)	+2.6% (+0.18 pp)
Pancake (Inline + Cache)	+6.5% (+0.09 pp)	+4.6% (+0.34 pp)

Table 6.4: Relative CPU utilisation overhead of the Ethernet driver with inline and global caching optimisations compared to C (Clang) baseline

We now examine how these driver-level improvements translate to system-wide performance. Figure 6.6 and Table 6.5 show throughput and CPU utilisation across the full echo server system. As expected, throughput remains identical across all implementations. The CPU utilisation gap narrows considerably: Pancake's system-wide overhead drops from +8.2% to +3.1% at low throughput, and from +4.2% to +1.8% at full throughput. Pancake now performs comparably to CompCert (+2.6% and +1.4% at low and full throughput respectively).

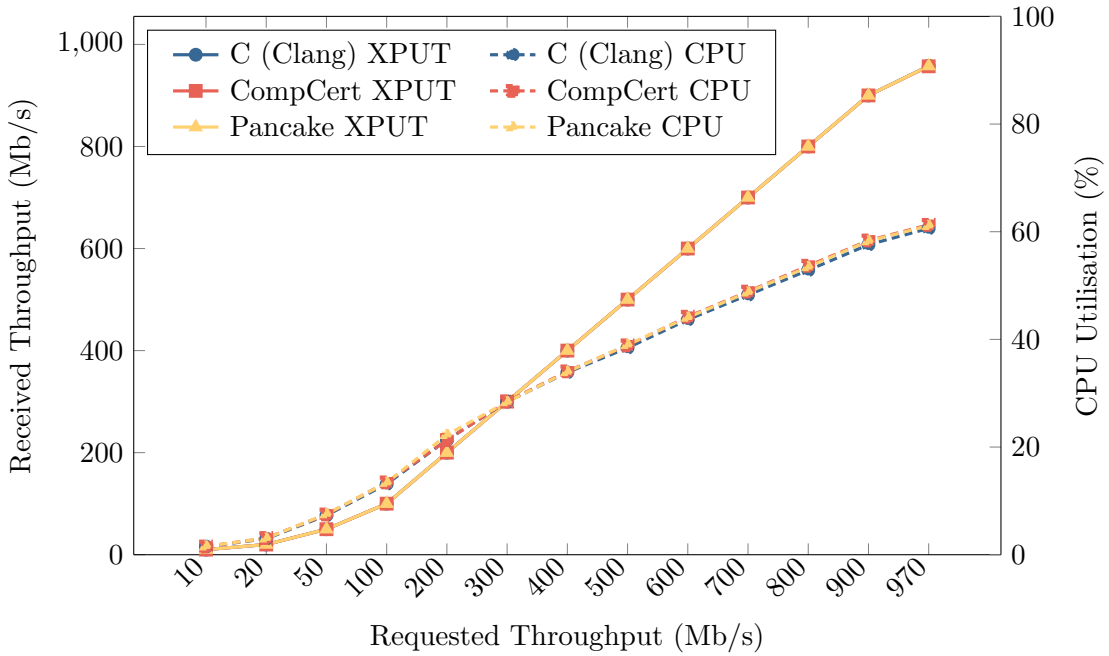

Figure 6.6: Throughput and CPU utilisation of the echo server with inline and global caching optimisations

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+2.6%	0.0%	+1.4%
Pancake (Inline + Cache)	0.0%	+3.1%	0.0%	+1.8%

Table 6.5: Relative difference in throughput and CPU utilisation with inline and global caching optimisations compared to C (Clang) baseline

Figure 6.7 confirms that the optimisations do not negatively impact latency. All three implementations exhibit comparable round-trip times across the load range. Notably, Pancake no longer shows the RTT elevation at 500 Mb/s observed in the baseline measurements, as its reduced CPU utilisation at this load now aligns with the other implementations.

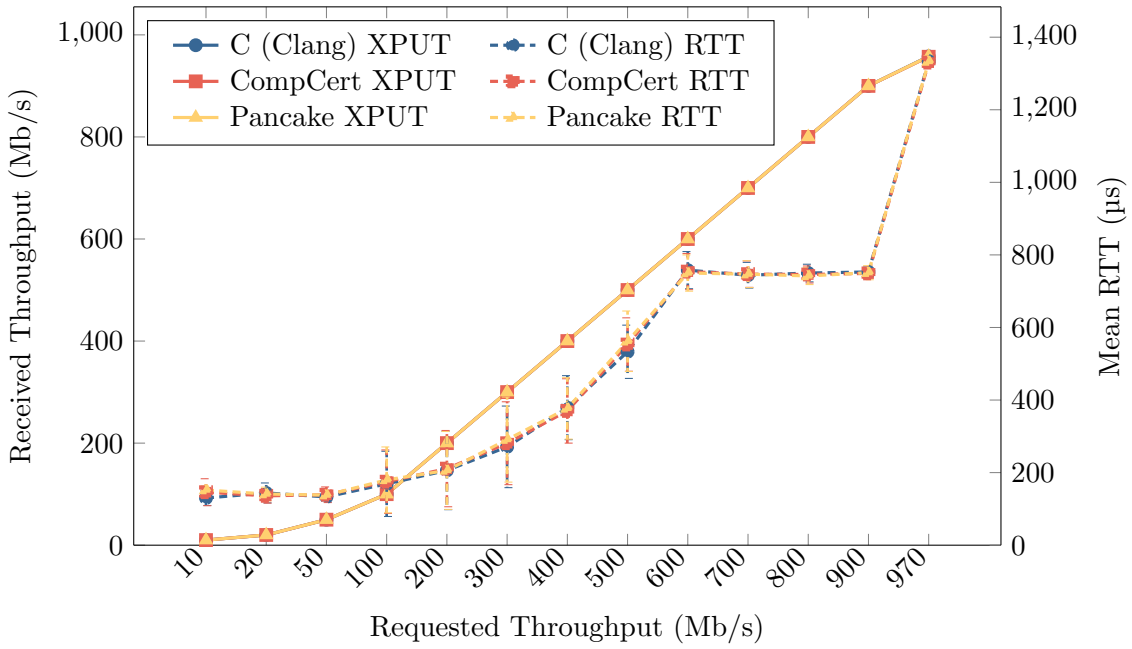

Figure 6.7: Throughput and mean round-trip time of the echo server with inline and global caching optimisations

6.2 Extending to the Full Network Stack

The previous sections focused on the Ethernet driver. We now extend our analysis to the complete LionsOS networking subsystem, which includes three additional components: the RX and TX virtualisers, responsible for multiplexing network traffic amongst clients, and the network copier, responsible for copying packet data between components.

We apply the same optimisation techniques from the previous sections (inlining and global variable caching) to these components. Additionally, we apply a known optimisation to the network copier: replacing the Pancake `memcpy` implementation with a foreign function call to the C standard library. This addresses a known performance bottleneck in the copier, where the naive Pancake byte-by-byte copy incurs significant overhead compared to the optimised C implementation.

6.2.1 Baseline Measurements

Figure 6.8 and Table 6.6 show throughput and CPU utilisation for the full network stack. As with previous experiments, all implementations achieve identical throughput, confirming that Pancake introduces no functional bottleneck. The system-wide CPU overhead for Pancake is +10.8% at low throughput and +8.1% at full throughput, while CompCert shows +3.4% and +4.5% respectively.

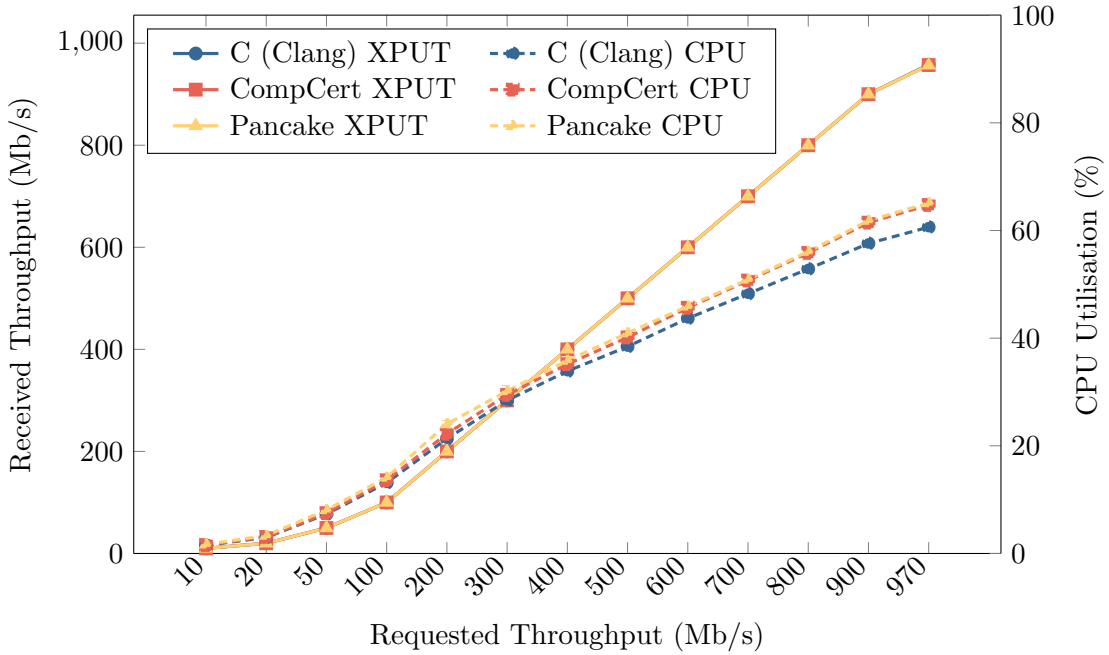

Figure 6.8: Throughput and CPU utilisation of the full network stack at varying requested throughput

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+3.4%	0.0%	+4.5%
Pancake	0.0%	+10.8%	0.0%	+8.1%

Table 6.6: Relative difference in throughput and CPU utilisation of the full network stack compared to C (Clang) baseline

To understand where this overhead originates, we examine each component individually. Figure 6.9, Figure 6.10, Figure 6.11, and Figure 6.12 show the per-component CPU utilisation, with Table 6.7 summarising the relative overhead. The RX virtualiser exhibits the highest overhead at +21.3% (low throughput) and +26.1% (full throughput), followed by the TX virtualiser at +17.0% and +14.5%. The Ethernet driver shows +10.2% and +5.9%, while the network copier shows the lowest overhead at +8.1% and +3.9%.

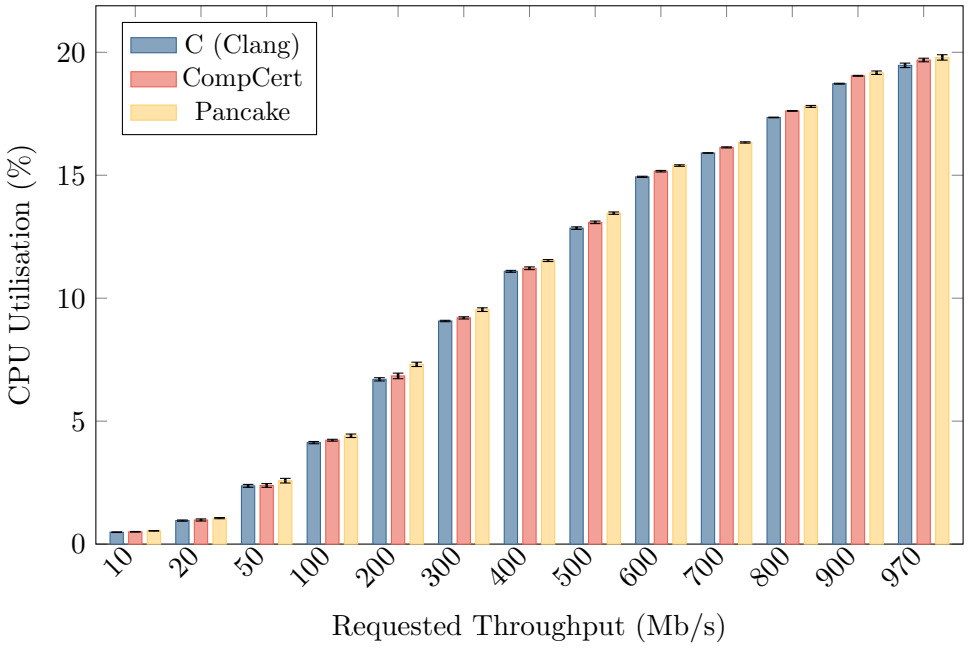

Figure 6.9: CPU utilisation of the Ethernet driver at varying requested throughput

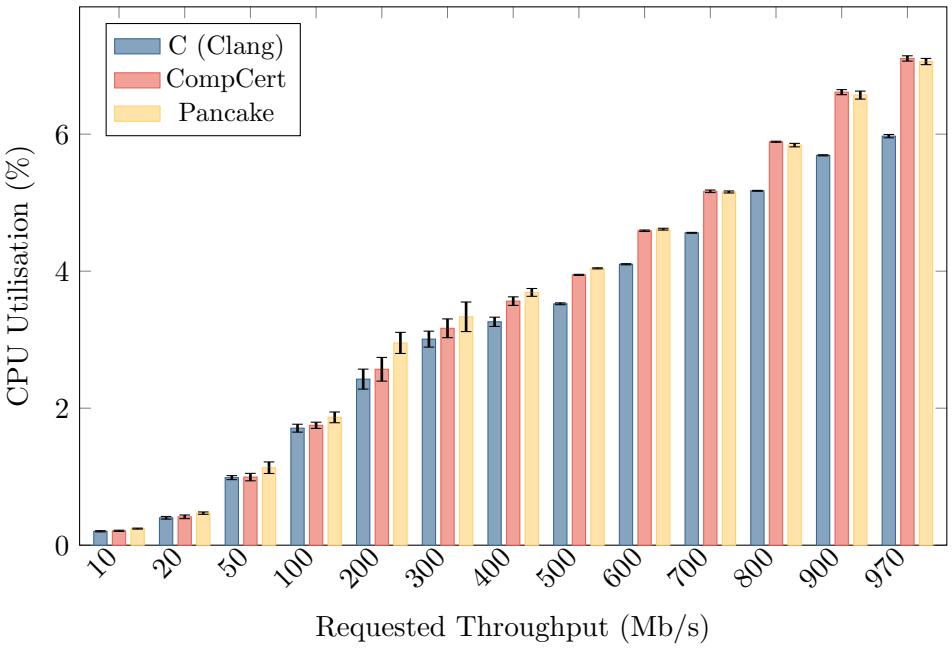

Figure 6.10: CPU utilisation of the TX virtualiser at varying requested throughput

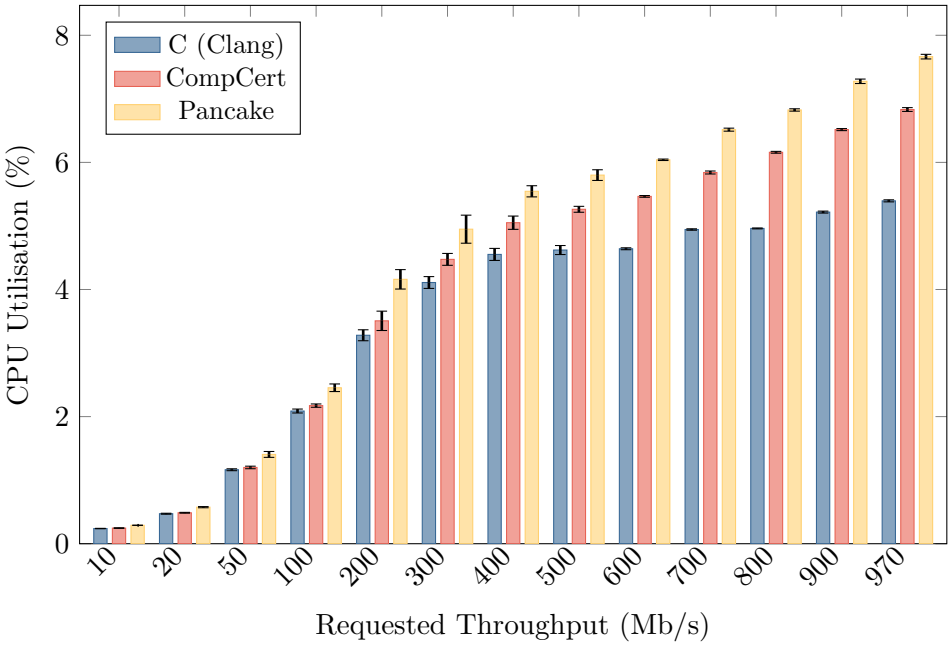

Figure 6.11: CPU utilisation of the RX virtualiser at varying requested throughput

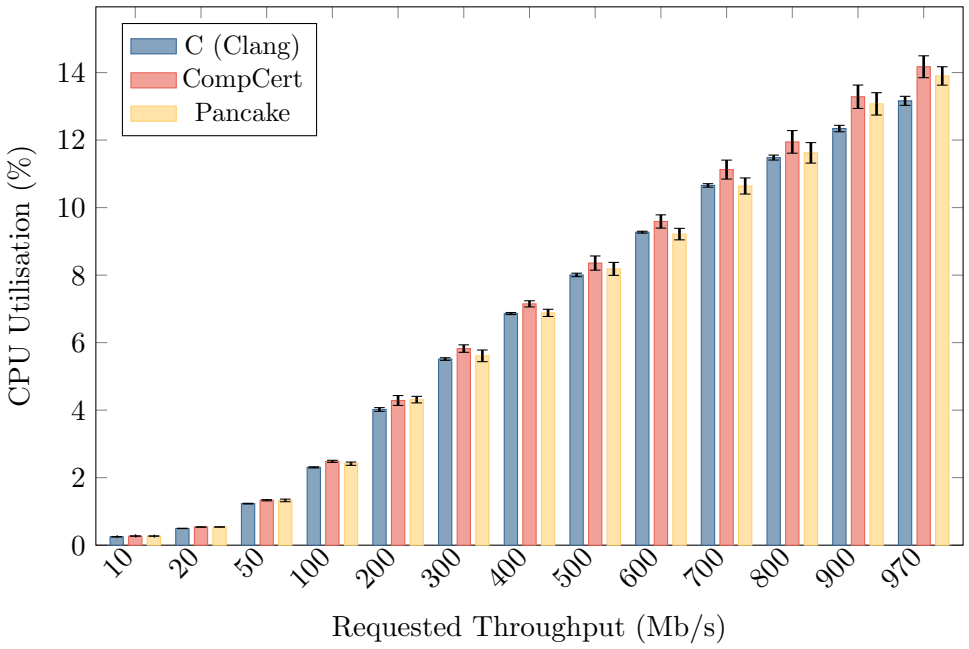

Figure 6.12: CPU utilisation of the network copier at varying requested throughput

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	CompCert	Pancake	CompCert	Pancake
Ethernet Driver	+2.2% (+0.02 pp)	+10.2% (+0.12 pp)	+1.7% (+0.16 pp)	+5.9% (+0.38 pp)
TX Virtualiser	+2.6% (+0.01 pp)	+17.0% (+0.08 pp)	+8.8% (+0.38 pp)	+14.5% (+0.41 pp)
RX Virtualiser	+3.1% (+0.02 pp)	+21.3% (+0.13 pp)	+12.4% (+0.58 pp)	+26.1% (+0.96 pp)
Network Copier	+8.2% (+0.05 pp)	+8.1% (+0.05 pp)	+6.2% (+0.37 pp)	+3.9% (+0.14 pp)

Table 6.7: Relative CPU utilisation overhead per component compared to C (Clang) baseline

We note that the CompCert Ethernet driver overhead in this configuration (+2.2% at low throughput, +1.7% at full throughput) is lower than in the driver-only measurements (+4.9% and +2.6% in Table 6.4). Network benchmarks involve complex interactions between components through shared queues and notification mechanisms, where changes in the performance characteristics of one component can affect the behaviour of others. Thus it is quite possible the introduction of other CompCert components may have changed the system behaviour and in turn affected the driver's performance characteristics. Given the scope of this thesis, and the fact that it was a positive improvement for CompCert, we do not investigate this further.

For Pancake, the virtualisers exhibit disproportionately high overhead compared to other components. This is despite the inlining and global variable caching optimisations already applied. Thus, we will investigate additional optimisation opportunities specific to the virtualisers in the following section.

6.2.2 Loop Unrolling

The virtualisers contain functions with small, fixed-iteration loops that execute frequently during packet processing. For example, the RX virtualiser's MAC address matching function in Listing 6.1 iterates over all clients and for each client iterates over 6 words to compare destination addresses.

```
fun get_mac_addr_match(1 buffer_vaddr) {
    var num_clients = pnk_mem(NUM_CLIENTS);
    var client = 0;
    while (client < num_clients) {
        var match = 1;
        var i = 0;
        while (i < 6) {
            var dest_byte = 0;
            var client_byte = pnk_mem(MAC_ADDR_BASE + client * 6 + i);
            var dest_addr = buffer_vaddr + i;
            !ld8 dest_byte, dest_addr;
            if (dest_byte != client_byte) {
                match = 0;
            }
            i = i + 1;
        }
        ...
    }
    ...
}
```

Listing 6.1: RX virtualiser MAC address matching function

Similarly, the TX virtualiser's offset extraction function in Listing 6.2 iterates over clients to determine which client a buffer belongs to.

```
fun extract_offset(1 phys_addr) {
    var num_clients = pnk_mem(NUM_CLIENTS);
    var client = 0;
    while (client < num_clients) {
        var cli_data_io = pnk_mem(CLI_DATA_IO_BASE + client);
        var cli_capacity = pnk_mem(CLI_QUEUE_BASE + client * 4 + 2);
        var cli_max_addr = cli_data_io + cli_capacity * 2048;
        if (phys_addr >= cli_data_io && phys_addr < cli_max_addr) {
            var offset = phys_addr - cli_data_io;
            return <client, offset>;
        }
        client = client + 1;
    }
    return < -1, 0 >;
}
```

Listing 6.2: TX virtualiser offset extraction function

In our system configuration, these loop bounds are constant and small: the inner loop always iterates exactly 6 times (the size of a MAC address), and the outer loop iterates once per client. We manually unroll both loops, eliminating per-iteration condition checks, increments, and repeated memory loads.

Figure 6.13 and Figure 6.14 show the virtualiser CPU utilisation comparing Pancake baseline against the optimised version with loop unrolling. As summarised in Table 6.8, the RX virtualiser overhead drops from +21.3% to +11.6% at low throughput and from +26.1% to +14.4% at full throughput. The TX virtualiser overhead reduces from +17.0% to +13.1% at low throughput and from +14.5% to +11.6% at full throughput. Figure 6.15 confirms that the Ethernet driver and network copier remain largely unchanged, as expected.

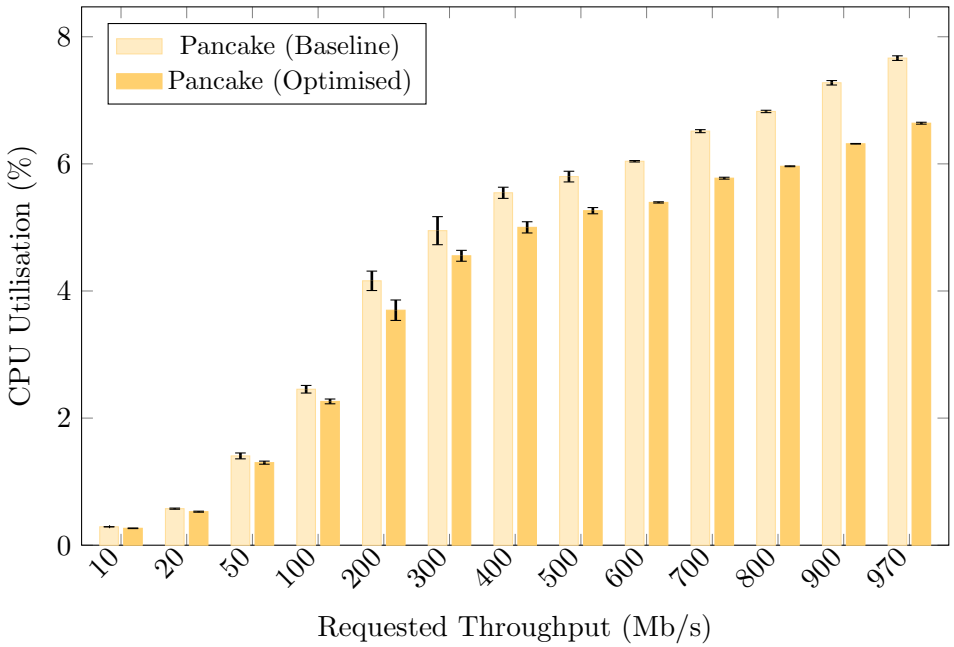

Figure 6.13: CPU utilisation of the RX virtualiser: baseline vs loop unrolling

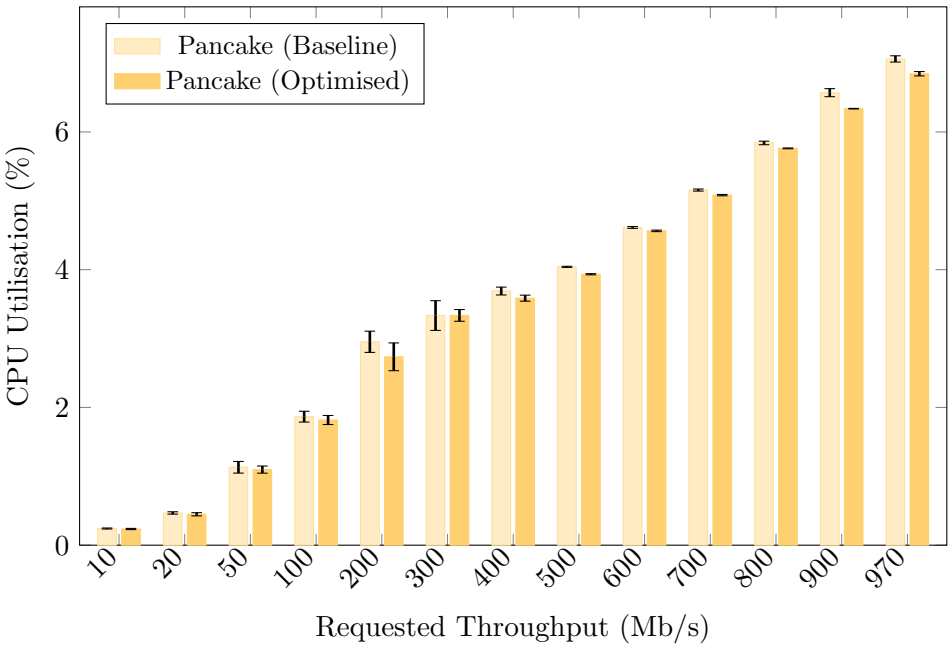

Figure 6.14: CPU utilisation of the TX virtualiser: baseline vs loop unrolling

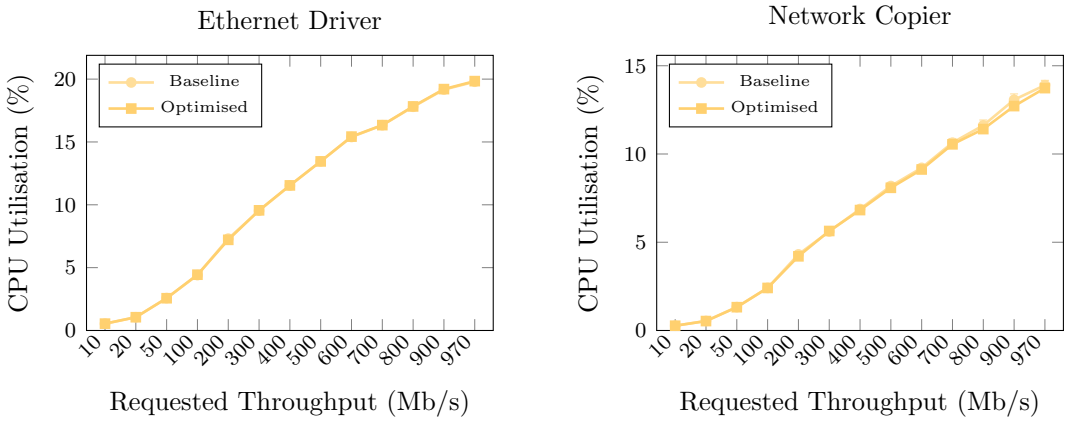

Figure 6.15: CPU utilisation of other components: baseline vs loop unrolling

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Baseline	Optimised	Baseline	Optimised
Ethernet Driver	+10.2% (+0.12 pp)	+10.4% (+0.12 pp)	+5.9% (+0.38 pp)	+5.6% (+0.38 pp)
TX Virtualiser	**+17.0% (+0.08 pp)**	**+13.1% (+0.06 pp)**	**+14.5% (+0.41 pp)**	**+11.6% (+0.37 pp)**
RX Virtualiser	**+21.3% (+0.13 pp)**	**+11.6% (+0.07 pp)**	**+26.1% (+0.96 pp)**	**+14.4% (+0.56 pp)**
Network Copier	+8.1% (+0.05 pp)	+7.6% (+0.05 pp)	+3.9% (+0.14 pp)	+2.9% (+0.09 pp)

Table 6.8: Pancake CPU utilisation overhead per component: baseline vs loop unrolling (relative to C)

We now examine how this optimisation translates to system-wide utilisation and performance in Figure 6.16 and Figure 6.17. Throughput remains identical, and round-trip time remains comparable across all implementations . As summarised in Table 6.9, the system-wide Pancake CPU overhead reduces from +10.8% to +8.7% at low throughput and from +8.1% to +5.9% at full throughput.

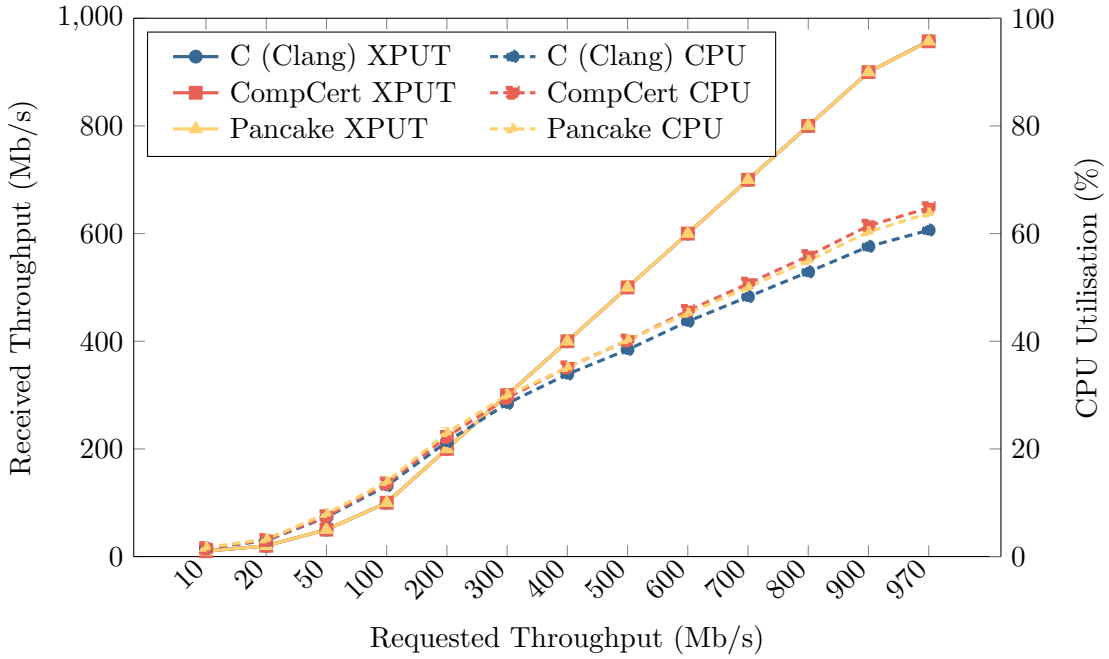

Figure 6.16: Throughput and CPU utilisation of the full network stack with loop unrolling applied

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+3.4%	0.0%	+4.5%
Pancake (Baseline)	0.0%	+10.8%	0.0%	+8.1%
Pancake (Optimised)	0.0%	+8.7%	0.0%	+5.9%

Table 6.9: System-wide CPU utilisation overhead with loop unrolling compared to C (Clang) baseline

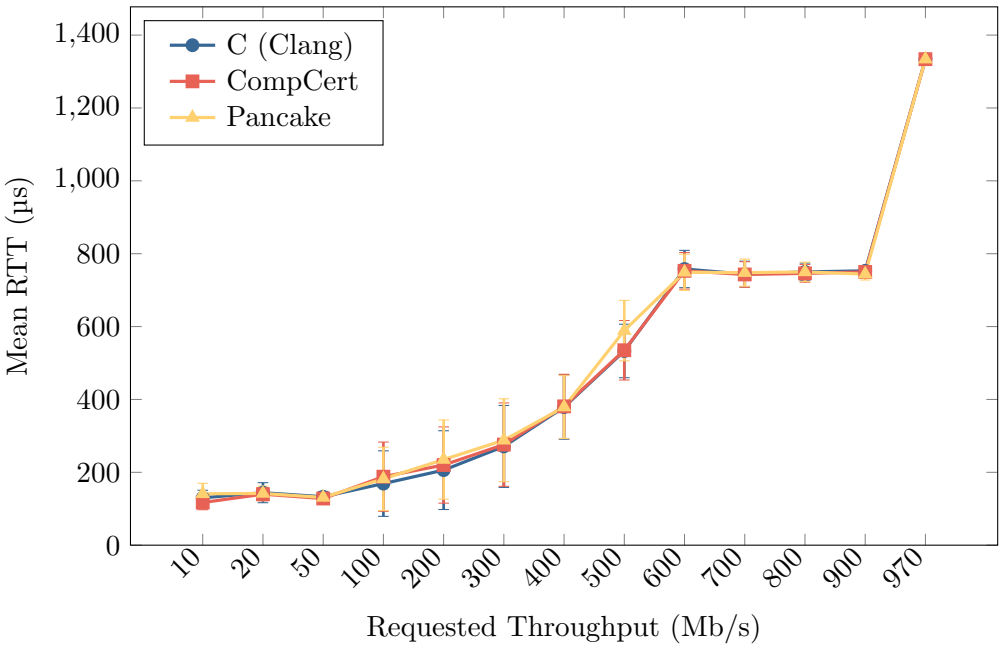

Figure 6.17: Mean round-trip time of the full network stack with loop unrolling applied

Thus far, we have shown that with loop unrolling, inlining, and global variable caching, the LionsOS networking subsystem implemented in Pancake achieves comparable system-wide performance to CompCert and remains within 10 to 15 percent overhead of C. Our results suggest that the remaining overhead is largely attributable to FFI costs, as each of the components evaluated requires an FFI call within its hot loop for memory barriers, cache management, or memory copying. We discuss this overhead further in Section 6.3.2.

This leaves us with one major component that remains implemented in C: the `libmicrokit` handler loop. In Section 6.3, we will evaluate and analyse its impact on system performance.

6.3 `libmicrokit` Performance

In this section we provide preliminary evaluation of the Pancake port of `libmicrokit`. We first present results for an echo server where only `libmicrokit`'s handler loop is implemented in Pancake while all other components remain in C compiled with Clang. We then evaluate a echo server system with its networking subsystem implemented entirely in Pancake, followed by an analysis of the source of performance overhead and initial evaluation of potential optimisation.

6.3.1 Baseline: Pancake `libmicrokit` with C Components

Figure 6.18 and Figure 6.19 present the throughput, CPU utilisation, and round-trip time measurements for the echo server using Pancake's `libmicrokit` port with all components in C. This configuration isolates the overhead introduced solely by the Pancake `libmicrokit`'s handler loop.

All three implementations achieve identical throughput across the entire load range, confirming that the Pancake `libmicrokit` port introduces no network performance regression. The CPU utilisation results in Table 6.10 show that CompCert incurs minimal overhead (+1.3% at low throughput, +1.6% at full throughput), while Pancake exhibits a much higher overhead (+10.1% at low throughput, +4.7% at full throughput).

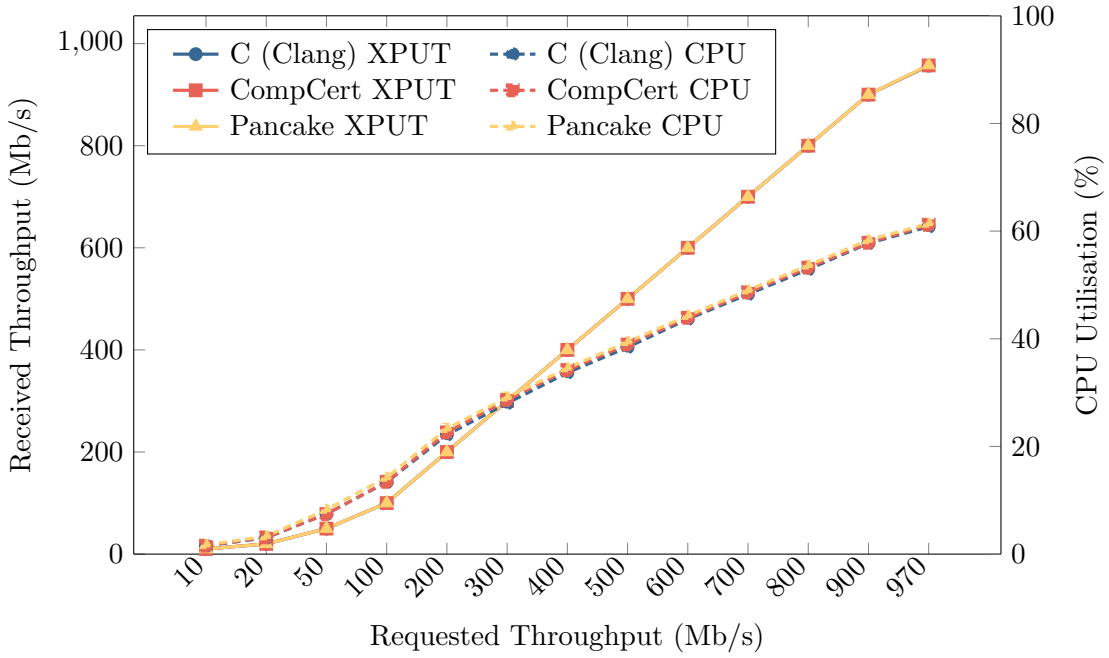

Figure 6.18: Throughput and CPU utilisation of the echo server with libmicrokit baseline

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+1.6%	0.0%	+1.3%
Pancake	0.0%	+10.1%	0.0%	+4.7%

Table 6.10: Relative difference in throughput and CPU utilisation with libmicrokit baseline compared to C (Clang)

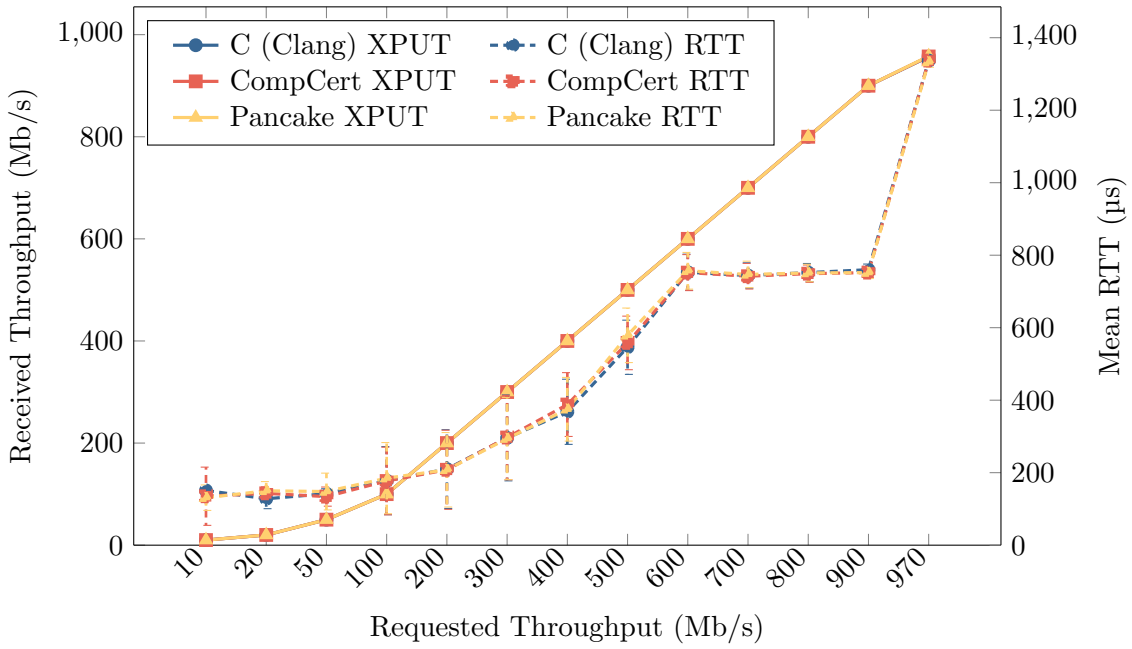

Figure 6.19: Throughput and mean round-trip time of the echo server with libmicrokit baseline

The per-component breakdown in Table 6.11 and Figure 6.20 reveals that the virtualiser components contribute the largest relative overhead for Pancake at low throughput, with the TX and RX virtualisers showing +20.9% and +16.7% overhead respectively. The network copier shows moderate overhead (+11.7%), while the echo client and ethernet driver show more modest overhead (+7.7% and +6.7% respectively). CompCert maintains consistently low overhead across all components, with the maximum observed overhead being 4.1% for the network copier.

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	CompCert	Pancake	CompCert	Pancake
Ethernet Driver	−0.2% (−0.01 pp)	+6.7% (+0.08 pp)	+0.2% (+0.03 pp)	+3.1% (+0.16 pp)
TX Virtualiser	+3.3% (+0.01 pp)	+20.9% (+0.11 pp)	+1.6% (+0.03 pp)	+9.0% (+0.14 pp)
RX Virtualiser	+2.9% (+0.01 pp)	+16.7% (+0.10 pp)	+1.9% (+0.05 pp)	+9.1% (+0.21 pp)
Echo Client	+1.5% (+0.00 pp)	+7.7% (+0.06 pp)	+1.2% (+0.09 pp)	+3.4% (+0.15 pp)
Network Copier	+4.1% (+0.03 pp)	+11.7% (+0.08 pp)	+2.2% (+0.08 pp)	+4.8% (+0.09 pp)

Table 6.11: Relative CPU utilisation overhead per component with `libmicrokit` baseline compared to C (Clang)

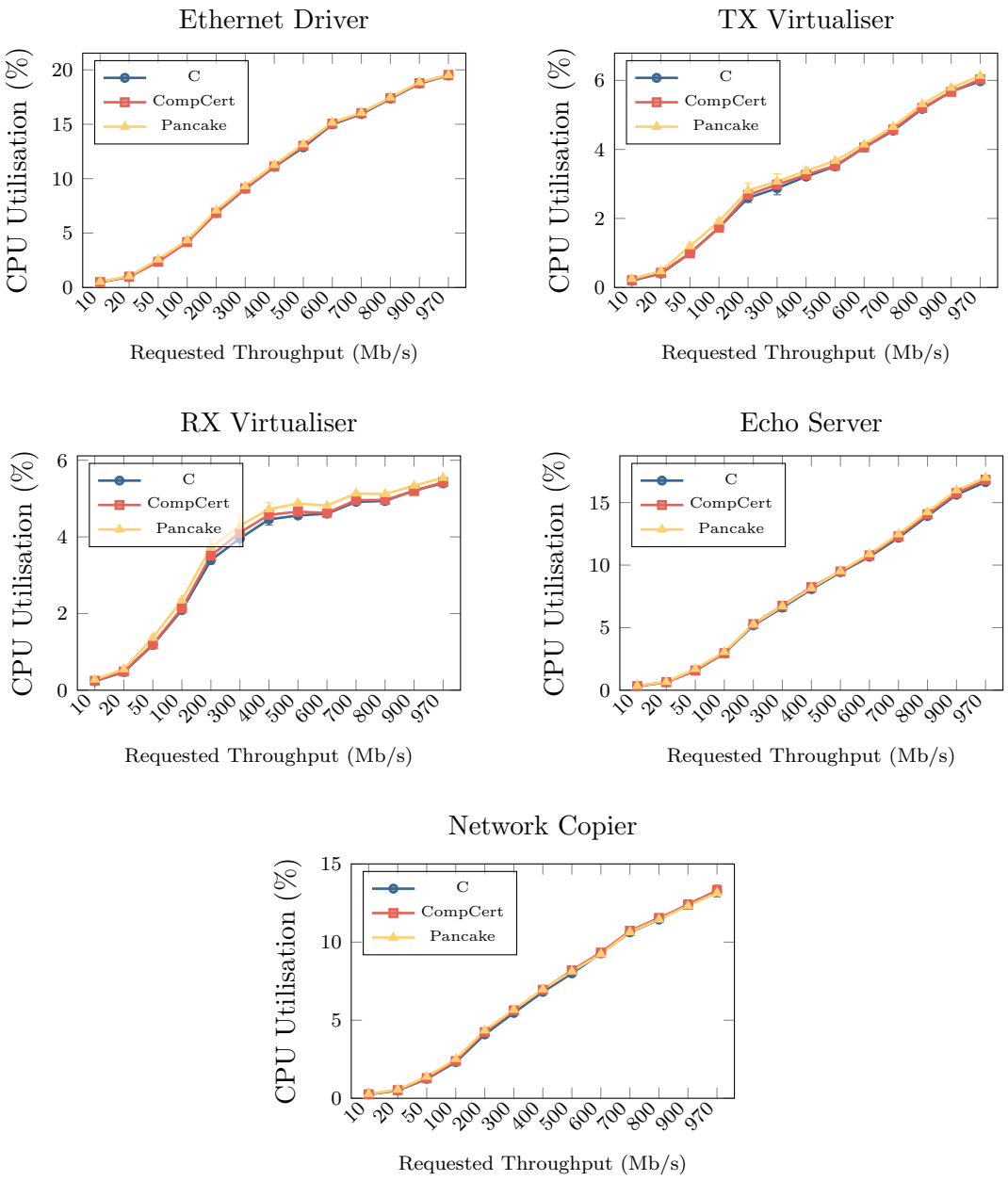

Figure 6.20: Per-component CPU utilisation of the echo server with libmicrokit baseline

The overhead we observe for the Pancake `libmicrokit` port in this section is substantial but expected, as the handler loop runs continuously during the benchmarks for all active components. We have now established two measurements: the overhead of an optimised Pancake network subsystem and the overhead of an unoptimised Pancake `libmicrokit` handler loop. A complete Pancake system (i.e. combining both components) would accumulate these overheads, so we should expect a notable increase in overall CPU utilisation compared to C. We now turn to this complete system.

6.3.2 Complete Pancake LionsOS Networking Subsystem

We now examine the complete Pancake LionsOS networking subsystem, formed by integrating the `libmicrokit` handler loop from Section 6.3.1 with the networking subsystem detailed in Section 6.2. As in earlier experiments, we compare against a baseline C system and a baseline CompCert system, each constructed using their respective networking subsystem components and the corresponding `libmicrokit` implementation.

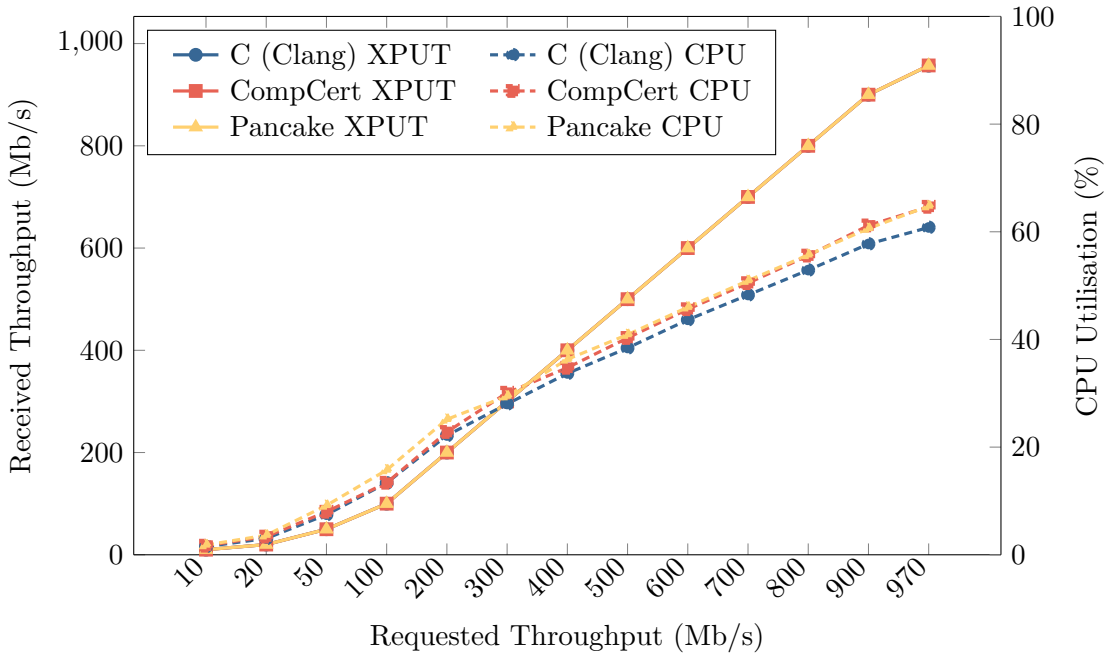

Figure 6.21: Throughput and CPU utilisation of the echo server with full network stack using libmicrokit

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+7.5%	0.0%	+5.8%
Pancake	0.0%	+20.2%	0.0%	+10.9%

Table 6.12: Relative difference in throughput and CPU utilisation with full network stack using `libmicrokit` compared to C (Clang) baseline

Figure 6.21 shows the system-wide throughput and CPU utilisation for the complete Pancake networking subsystem. As summarised in Table 6.12, all implementations achieve identical throughput. However, we observe a notable increase in CPU overhead compared to our earlier measurements. Pancake exhibits +20.2% overhead at low throughput and +10.9% at full throughput, compared to +8.7% and +5.9% respectively when using the C `libmicrokit` handler loop (Table 6.9). CompCert overhead

also increases from +3.4%/+4.5% to +7.5%/+5.8%. These results confirm that combining the Pancake `libmicrokit` handler loop with the Pancake networking components compounds the overhead from both sources.

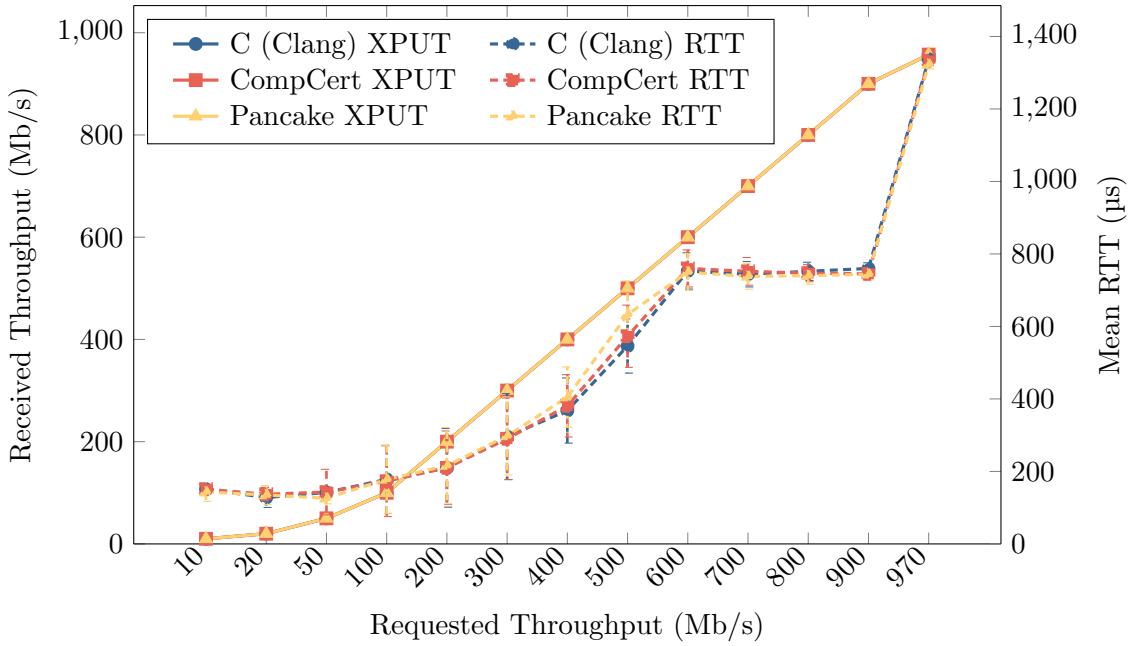

Figure 6.22: Throughput and mean round-trip time of the echo server with full network stack using libmicrokit

Figure 6.22 shows that just like the earlier experiments, round-trip time remains comparable across all implementations, indicating that the increased CPU overhead is not significant enough to translate to user-visible latency degradation.

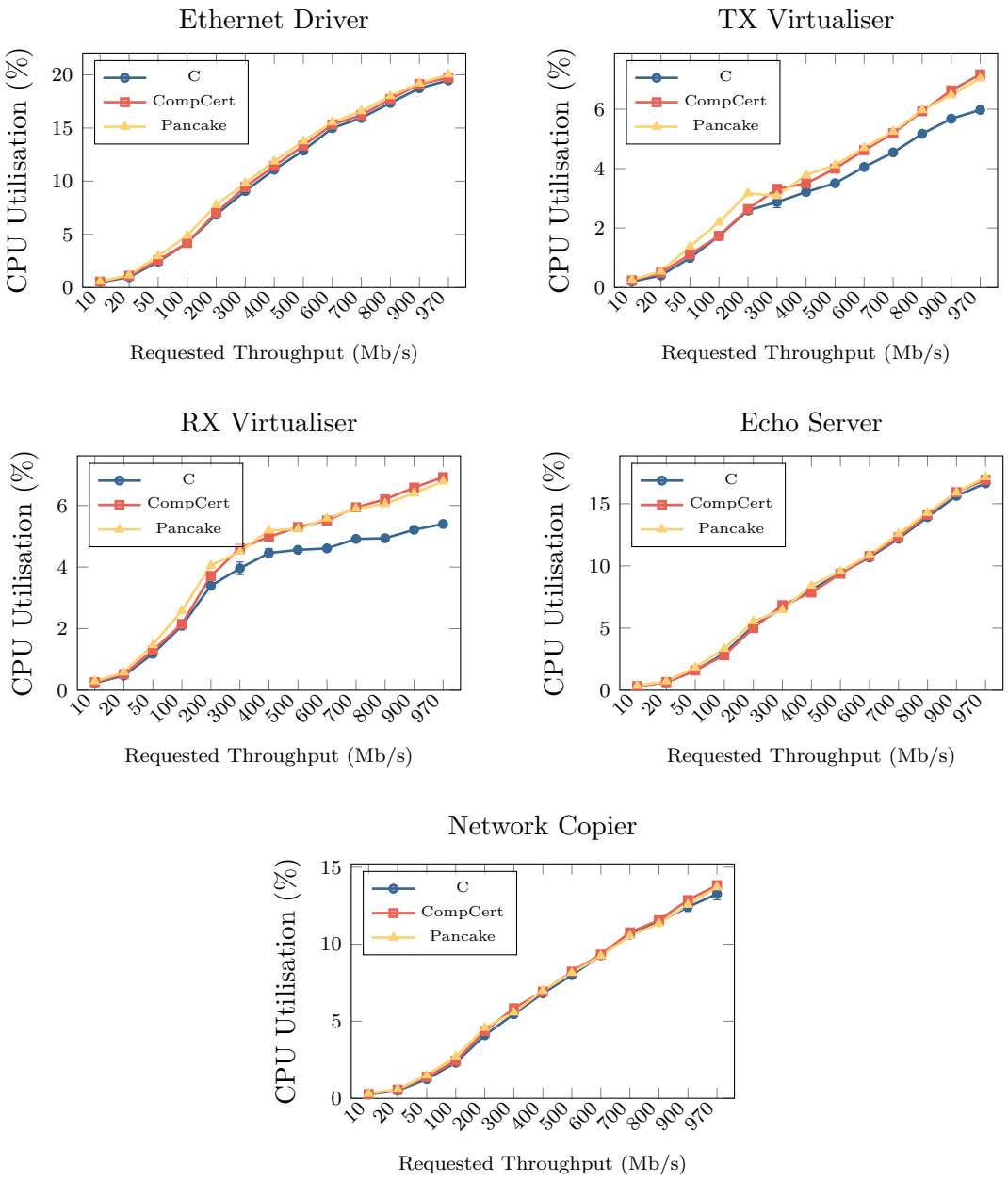

Figure 6.23: Per-component CPU utilisation of the echo server with full network stack using libmicrokit

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	CompCert	Pancake	CompCert	Pancake
Ethernet Driver	+10.1% (+0.11 pp)	+18.6% (+0.27 pp)	+4.1% (+0.25 pp)	+9.6% (+0.59 pp)
TX Virtualiser	+18.5% (+0.08 pp)	+34.2% (+0.19 pp)	+13.5% (+0.43 pp)	+21.0% (+0.54 pp)
RX Virtualiser	+9.7% (+0.06 pp)	+23.2% (+0.15 pp)	+15.9% (+0.66 pp)	+20.7% (+0.71 pp)
Echo Server	+2.2% (+0.01 pp)	+13.8% (+0.12 pp)	+0.4% (+0.05 pp)	+5.9% (+0.24 pp)
Network Copier	+12.3% (+0.08 pp)	+19.6% (+0.13 pp)	+5.4% (+0.21 pp)	+7.4% (+0.16 pp)

Table 6.13: Relative CPU utilisation overhead per component with full network stack using libmicrokit compared to C (Clang)

Figure 6.23 and Table 6.13 show the per-component CPU utilisation. The results match our expectations: each component exhibits an overhead approximately equal to the sum of its individual overhead from Table 6.8 and the `libmicrokit` handler loop overhead described in Section 6.3.1. Having applied best-effort optimisations to the networking components, we now shift our focus to understanding the sources of the remaining overhead in `libmicrokit`.

Where does the overhead come from?

An immediate observation is that the `libmicrokit` event loop executes continuously throughout the lifetime of the system, which amplifies even minor inefficiencies and turns small costs into noticeable overhead. To understand its behaviour in more detail, we now examine the generated code of the Pancake event loop. From this inspection, we will hypothesise two factors that likely contribute to the remaining overhead, which we outline below.

FFI call overhead and memory access patterns Recall from Section 5.2.7 that Pancake's FFI mechanism does not support return values, requiring workarounds such as writing results to shared memory locations. In the `libmicrokit` handler loop, this limitation compounds with the lack of inline assembly support to create significant overhead.

Since Pancake cannot directly invoke seL4 system calls, we must call out to C wrapper functions via FFI. Each FFI call requires explicit argument setup, manual return address management, and register spilling. Listing 6.3 shows the assembly generated for a single FFI call to check for pending signals.

```
; @microkit_have_signal(0, HAVE_SIGNAL_ADDR, 0, 0)
mov     x2, #0x0            ; Setup argument
mov     x1, #0x1f6          ; HAVE_SIGNAL_ADDR = 502
str     x0, [x25, #8]       ; Spill have_reply to stack
orr     x0, x2, x2          ; x0 = 0
orr     x3, x2, x2          ; x3 = 0
str     x30, [x25, #32]     ; Save return address
```

```
adr     x30, .Lreturn       ; Set manual return address
b       cake_ffimicrokit_have_signal
```

Listing 6.3: Pancake-generated FFI call setup

After each FFI call returns, Pancake must reload spilled values and shuffle registers. Listing 6.4 shows this overhead.

```
.Lreturn:
ldr     x24, [x25, #32]     ; Reload saved value
str     x24, [x25, #24]     ; Shuffle to different slot
orr     x30, x28, x28       ; Copy base pointer
ldr     x30, [x30, #4016]   ; Load microkit_have_signal result
ldr     x24, [x25, #8]      ; Reload have_reply
str     x24, [x25, #16]     ; Shuffle to another slot
```

Listing 6.4: Pancake-generated post-FFI register restoration

In contrast, an equivalent C implementation would compile to a single `bl` (branch-and-link) instruction, with variables remaining in registers throughout. The need to synchronise variables through memory also affects code outside of FFI calls. Listing 6.5 shows `have_reply` being loaded from memory for a simple conditional check, whereas C would keep this value in a register.

```
; if (have_reply) - variable loaded from memory
ldr     x24, [x25, #8]      ; Load have_reply from stack
cmp     x24, #0x0           ; Compare
b.ne    .Lreply_branch      ; Branch if true
```

Listing 6.5: Pancake-generated variable access from memory

The current design also introduces unnecessary indirection for notification and protected procedure call handlers. When the handler loop receives a notification, it FFIs into a C wrapper function (e.g., `ffimicrokit_notified_handler`), which calls the user's Pancake handler as an external function. When that Pancake handler needs to perform seL4 system calls, such as notifying another component, it must FFI back into C. This Pancake-to-C-to-Pancake-to-C indirection adds overhead at each transition.

This indirection also impacts instruction cache performance. The repeated transitions between Pancake and C code fragments results in poor code locality, as execution jumps between disparate memory regions. Figure 6.24 shows the L1 instruction cache misses per packet across the throughput range. Pancake exhibits +42.9% more L1 instruction cache misses than C on average, while CompCert shows +12.7%. This cache pressure contributes to the overall performance overhead observed in the handler loop.

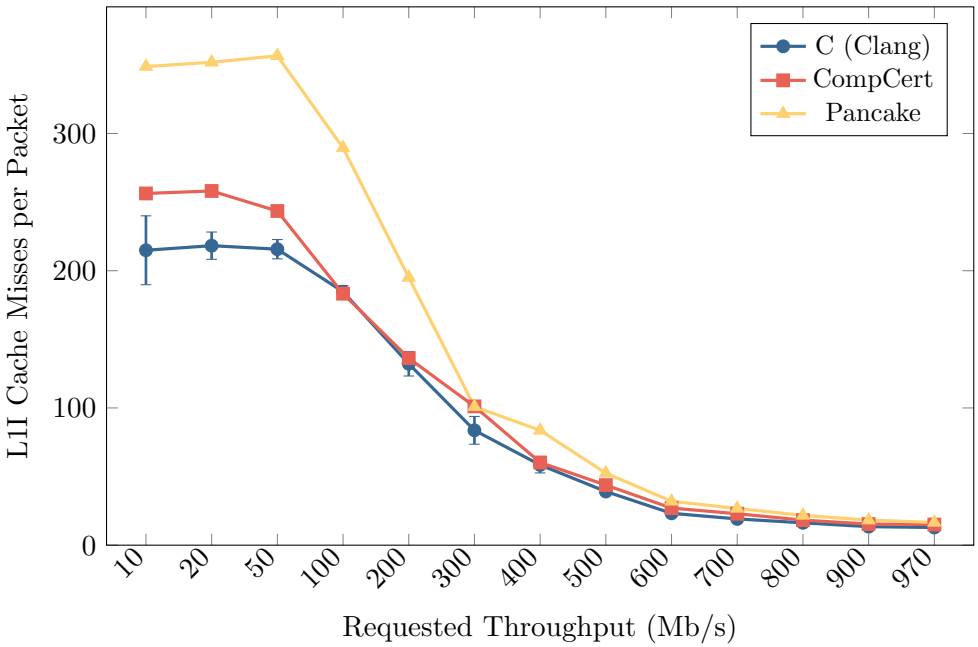

Figure 6.24: L1 instruction cache misses per packet for the full network stack with `libmicrokit`

The solution to these issues is to add support for inline assembly in Pancake. This would allow system calls to be invoked directly, eliminating FFI call setup and restoration overhead. Variables could remain in registers across system call boundaries, removing the need for memory synchronisation. The notification and protected procedure call handlers could invoke system calls directly without intermediate C trampolines, improving code locality and reducing instruction cache pressure. Implementing this feature is beyond the scope of this thesis but represents a clear path to reducing the `libmicrokit` handler loop overhead.

Linear notification scan The original Pancake handler loop implementation uses a linear scan to process notification badges. Listing 6.6 shows the original notification handling code.

```
var idx = 0;
while (badge != 0) {
    if (badge & 1) {
        @microkit_notified_handler(0, idx, 0, 0);
    }
    badge = badge >> 1;
    idx = idx + 1;
}
```

Listing 6.6: Original Pancake linear notification scan

This implementation iterates through all 64 bits of the badge, checking each bit individually and calling the notification handler when a bit is set. For sparse badges where only a few bits are set, this wastes cycles on empty iterations. In contrast, C compilers optimise an equivalent loop using hardware bit manipulation instructions to directly find set bits. Listing 6.7 shows the equivalent C code using `__builtin_ctzll` (count trailing zeros) to jump directly to set bits.

```
while (badge) {
    int idx = __builtin_ctzll(badge);  // Find lowest set bit
    notified(idx);
    badge &= badge - 1;                // Clear lowest set bit
}
```

Listing 6.7: Optimised notification scan using bit manipulation

We replace the Pancake linear scan with an FFI call to a C function that follows this optimised pattern. In our current echo server configuration, this optimisation has minimal impact because each component has at most 3 notification channels, so the linear scan terminates early regardless. However, in more complex systems with many inter-component communication channels, such as a firewall[1] with multiple network interfaces and client connections, a component may need to scan a significant portion of the 64-bit badge. In such scenarios, the linear scan overhead would become more pronounced.

Figure 6.25 shows the system-wide throughput and CPU utilisation with the notification scan optimisation applied. As summarised in Table 6.14, Pancake overhead reduces from +20.2% to +17.1% at low throughput and from +10.9% to +10.4% at full throughput. This modest improvement confirms that the linear scan is not the dominant source of overhead in our current configuration, but the optimisation remains worthwhile for more complex system topologies.

[1] `https://lionsos.org/docs/examples/firewall/`

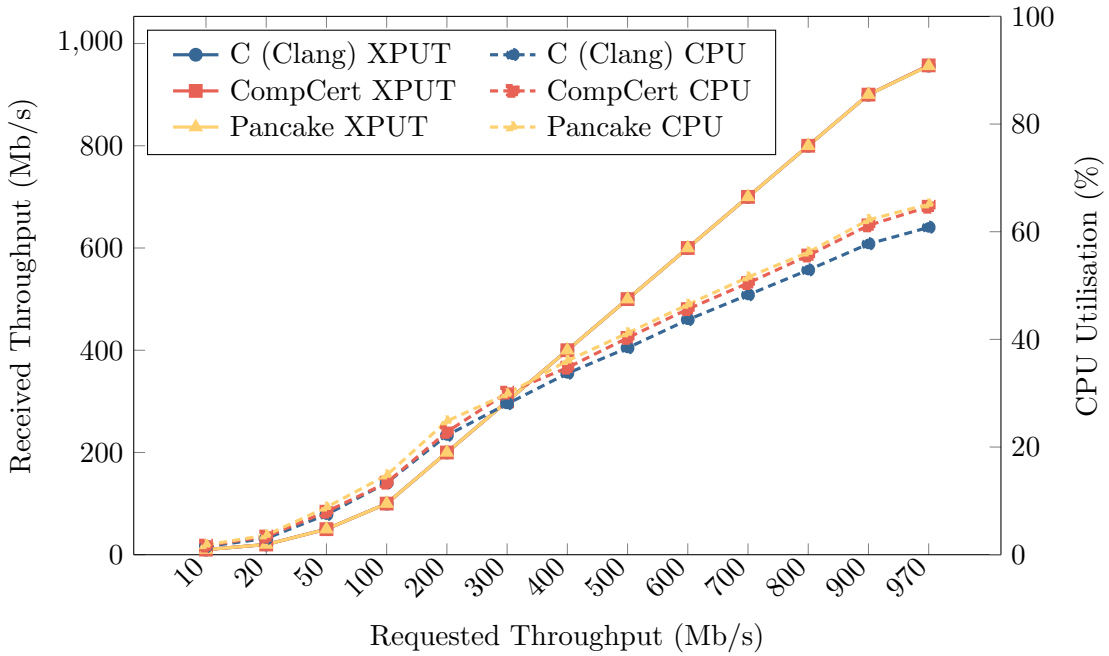

Figure 6.25: Throughput and CPU utilisation of the echo server with full network stack and notification optimisations

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	Throughput	CPU Util	Throughput	CPU Util
CompCert	0.0%	+7.5%	0.0%	+5.8%
Pancake (Optimised)	0.0%	+17.1%	0.0%	+10.4%

Table 6.14: Relative difference in throughput and CPU utilisation with full network stack and notification optimisations compared to C (Clang) baseline

Given that we observe a reduction in CPU utilisation, we do not expect any regression in latency performance. As expected, Figure 6.26 confirms that round-trip time remains comparable across all implementations.

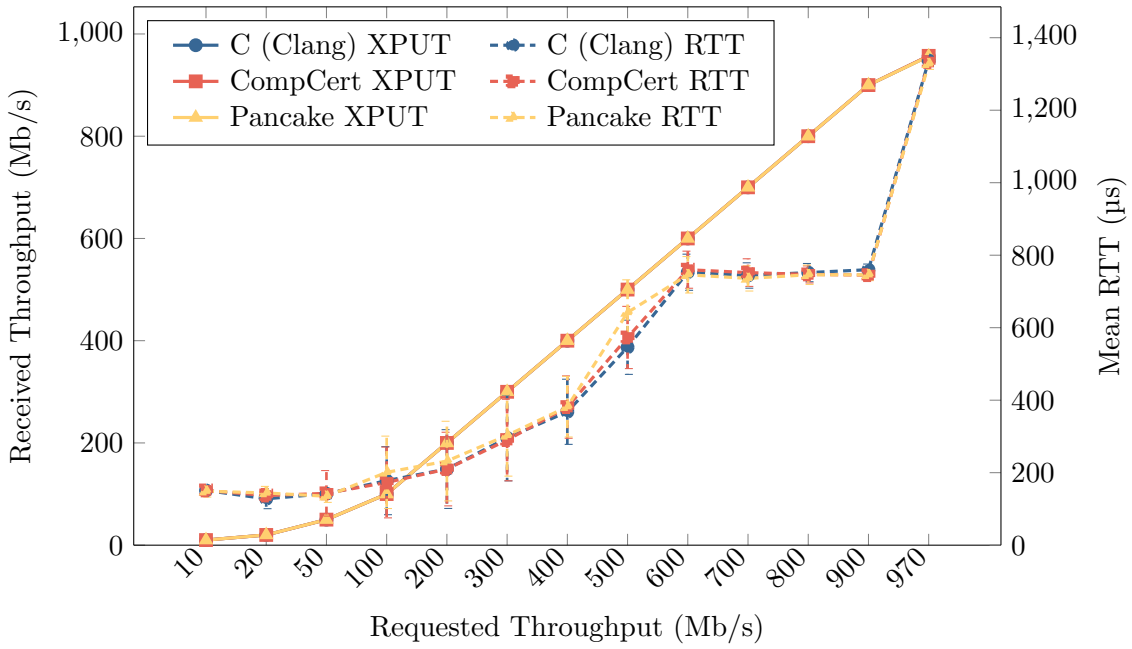

Figure 6.26: Throughput and mean round-trip time of the echo server with full network stack and notification optimisations

Figure 6.27 and Table 6.15 show the per-component CPU utilisation with the notification scan optimisation. We observe a consistent reduction in overhead across all components as expected.

	Low Throughput (10–100 Mb/s)		**Full Throughput (10–970 Mb/s)**	
	CompCert	Pancake	CompCert	Pancake
Ethernet Driver	+10.1% (+0.11 pp)	+17.6% (+0.23 pp)	+4.1% (+0.25 pp)	+8.6% (+0.52 pp)
TX Virtualiser	+18.5% (+0.08 pp)	+32.0% (+0.16 pp)	+13.5% (+0.43 pp)	+19.3% (+0.51 pp)
RX Virtualiser	+9.7% (+0.06 pp)	+21.8% (+0.13 pp)	+15.9% (+0.66 pp)	+19.5% (+0.68 pp)
Echo Server	+2.2% (+0.01 pp)	+13.6% (+0.11 pp)	+0.4% (+0.05 pp)	+5.2% (+0.21 pp)
Network Copier	+12.3% (+0.08 pp)	+17.0% (+0.11 pp)	+5.4% (+0.21 pp)	+9.6% (+0.49 pp)

Table 6.15: Relative CPU utilisation overhead per component with full network stack and notification optimisations compared to C (Clang)

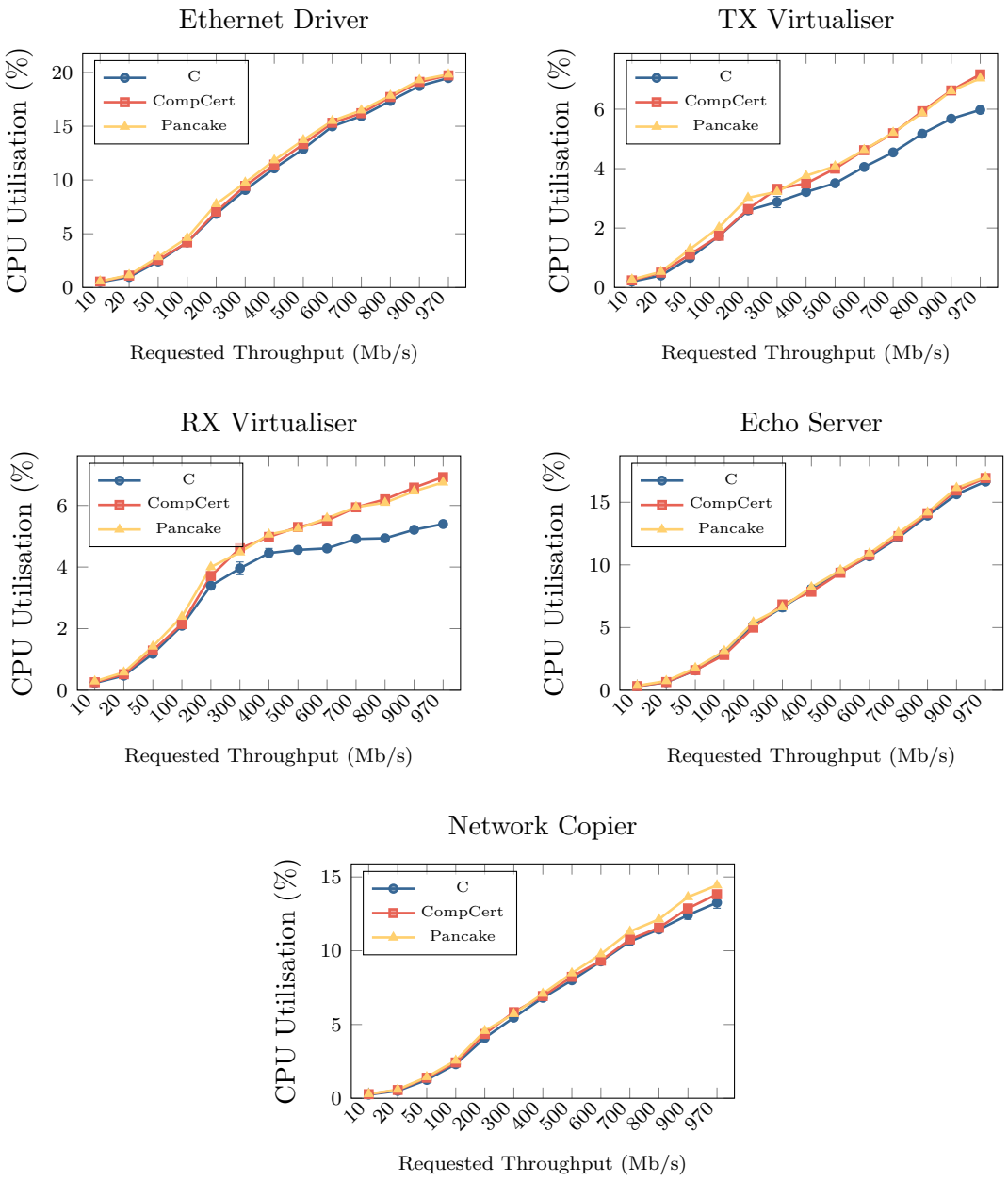

Figure 6.27: Per-component CPU utilisation of the echo server with full network stack and notification optimisations

Given the simplistic nature of the handler loop and our experience optimising Pancake code in the networking subsystem, we believe that addressing the limitations outlined in this section would significantly improve the performance of Pancake `libmicrokit`. In particular, adding inline assembly support to Pancake would eliminate the FFI overhead, remove the need for memory-based variable synchronisation, and improve code locality. These improvements would bring the Pancake handler loop performance closer to the C baseline.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis set out to evaluate the suitability of Pancake as a systems programming language for implementing LionsOS components. We approached this question by porting a representative suite of components, documenting the engineering effort involved, and measuring the performance of the resulting implementations.

We ported 20 LionsOS components to Pancake, comprising over 5,500 lines of code. This includes the majority of the seL4 Device Driver Framework, spanning Ethernet, serial, timer, and I^2C drivers across multiple hardware platforms, as well as network and serial virtualisers and the core `libmicrokit` event loop. These ports demonstrate that Pancake can express the low level systems code required for OS components and provide a foundation for future verification efforts.

Our engineering evaluation revealed the absence of major limitations in Pancake's expressiveness for systems programming. The average code expansion of 66% compared to C reflects the absence of high level abstractions such as named structures and the need for explicit memory management. Development time varied significantly across components, with the majority of effort spent on debugging rather than writing code. The quality of compiler error messages proved to be a significant obstacle, often providing minimal diagnostic information that made isolating errors tedious. We also encountered a compiler bug in the multiple entry point feature that required considerable effort to identify and work around.

Our performance evaluation compared Pancake implementations against C compiled with Clang and the verified CompCert compiler. Through targeted optimisations guided by our evaluation and analysis of the generated code, we brought our Pancake components within 10 to 15 percent of their C counterparts. This overhead is acceptable for practical deployment in performance sensitive systems and compares favourably with other verified compilation approaches.

We conclude that Pancake is a viable language for building LionsOS components. While the compiler requires further maturation, particularly in error reporting and debugging support, the core language is expressive enough for systems programming and the performance overhead is manageable. The ports we have produced provide a starting point for formal verification of LionsOS components, bringing the goal of a fully verified operating system stack closer to realisation.

7.2 Future Work

Several directions remain for future investigation.

7.2.1 Performance Optimisation

While we achieved our target of 10 to 15 percent overhead compared to C, further optimisation remains possible. In particular, our `libmicrokit` experiments revealed significant FFI overhead in the core event loop. Reducing this overhead, whether through compiler improvements or restructuring the FFI boundary, would benefit all LionsOS components.

7.2.2 Cross Platform Evaluation

Our performance evaluation focused on the ARM platform. Extending this evaluation to other architectures such as RISC-V and x86 would provide a more complete picture of Pancake's performance characteristics. Different architectures may exhibit different overhead profiles due to variations in calling conventions, register allocation, and instruction selection.

7.2.3 C to Pancake Transpiler

Porting C code to Pancake remains a manual and time consuming process. As discussed in Section 5.3, a production grade `C2Pancake` transpiler that provides semi-automated translation would significantly lower the barrier to adoption. Such a tool could handle straightforward syntactic transformations automatically while flagging constructs that require manual intervention, enabling faster porting of existing codebases and encouraging broader community engagement with the language.

7.2.4 Application Level Components

Our work focused on low level OS infrastructure. A natural next step is to explore application level components written in Pancake, such as a web server or network stack. These would exercise different aspects of the language and demonstrate Pancake's suitability for higher level systems code.

7.2.5 Verification

The components we have implemented provide a foundation for future verification efforts. Verifying the functional correctness of these Pancake components would demonstrate end to end verified compilation for LionsOS and bring the goal of a fully verified operating system stack closer to realisation.

7.2.6 Compiler Improvements

Our experience identified several areas where the Pancake compiler could be improved. Better error messages with accurate line numbers would significantly reduce debugging time. Native support for string literals or a debug printing facility would simplify runtime debugging. Addressing these issues would improve the developer experience and encourage wider adoption of the language.

Bibliography

Abrahamsson, O., Myreen, M. O., Kumar, R., and Sewell, T. (2022). Candle: A Verified Implementation of HOL Light. In Andronick, J. and de Moura, L., editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:17, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. (2016). Cogent: Verifying high-assurance file system implementations. In *ASPLOS*, pages 175–188, Atlanta, GA, USA.

Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., and Summers, A. J. (2022). The Prusti project: Formal verification for rust. In *NASA Formal Methods (14th International Symposium)*, pages 88–108. Springer.

Avnet Manufacturing Services (2019). Maaxboard hardware user manual v1.2. Development board based on NXP i.MX8M SoC.

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Brunel, J.-C., Moy, Y., Monate, B., Kanig, J., and Hubert, L. (2015). SPARK: Modular deductive verification for Ada. In *Proceedings of the 14th International Conference on Formal Methods and Software Engineering (FMSE)*, pages 392–398. Springer.

Butler, R. W. and Finelli, G. B. (1993). The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12.

Cebeci, C., Zou, Y., Zhou, D., Candea, G., and Pit-Claudel, C. (2024). Practical verification of system-software components written in standard C. In *ACM Symposium on Operating Systems Principles*. ACM.

Chen, H., Shanghai, Miao, X., Jia, N., Wang, N., Li, Y., Liu, N., Liu, Y., Wang, F., Huang, Q., Li, K., Yang, H., Wang, H., Yin, J., Peng, Y., , and Xu, F. (2024). Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *OSDI*, pages 465–485, Santa Clara, CA, US. Usenix.

Chen, X., Li, Z., Mesicek, L., Narayanan, V., and Burtsev, A. (2023). Atmosphere: Towards practical verified kernels in Rust. In *Workshop on Kernel Isolation, Safety and Verification*. ACM.

Cofer, D., Gacek, A., Backes, J., Whalen, M., Pike, L., Foltzer, A., Podhradsky, M., Klein, G., Kuz, I., Andronick, J., Heiser, G., and Stuart, D. (2018). A formal approach to constructing secure air vehicle software. *IEEE Comp.*, 51:14–23.

Dennis, J. B. and Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155.

Elkaduwe, D., Derrin, P., and Elphinstone, K. (2008). Kernel design for isolation and assurance of physical memory. In *WS Isolation & Integration Emb. Syst.*, pages 35–40, Glasgow, UK. ACM.

Feiertag, R. J. and Neumann, P. G. (1979). The foundations of a provably secure operating system (psos). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press.

FramaC (2008). Frama-C. `https://frama-c.com/`.

Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. (2016). CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, pages 653–669, Savannah, GA, US. USENIX Association.

Heiser, G. (2020). The seL4 microkernel – an introduction. seL4 Foundation Whitepaper.

Heiser, G. (2024). COMP9242 Advanced Operating Systems: Microkernel Design & Implementation – The 25-Year Quest for the Right API (Week 05 Part 1). `https://cgi.cse.unsw.edu.au/~cs9242/24/lectures/05a-uk.pdf`. Lecture slides, UNSW School of Computer Science & Engineering.

Heiser, G., Chubb, P., Brown, A., Darville, C., and Parker, L. (2024). sDDF design: Design, implementation and evaluation of the seL4 device driver framework.

Heiser, G., Parker, L., Chubb, P., Velickovic, I., and Leslie, B. (2022). Can we put the "S" into IoT? In *WFIoT*, Yokohama, JP.

Heiser, G., Velickovic, I., Chubb, P., Joshy, A., Ganesh, A., Nguyen, B., Li, C., Darville, C., Zhu, G., Archer, J., Zhou, J., Winter, K., Parker, L., Duchniewicz, S., and Bai, T. (2025). Fast, secure, adaptable: LionsOS design, implementation and performance. *arXiv preprint*. `https://arxiv.org/abs/2501.06234`.

Holt, R. C. (1982a). A short introduction to concurrent euclid. *SIGPLAN Not.*, 17(5):60–79.

Holt, R. C. (1982b). Tunis: a unix look-alike written in concurrent euclid (abstract). *SIGOPS Oper. Syst. Rev.*, 16(1):4–5.

Immunant, Inc. (2024). C2Rust: Migrate C code to Rust. `https://github.com/immunant/c2rust`. Accessed: November 2024.

Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2018). RustBelt: Securing the foundations of the Rust programming language. In *POPL*, pages 66:1–66:34.

Kanabar, H., Vivien, S., Abrahamsson, O., Myreen, M. O., Norrish, M., Pohjola, J. Å., and Zanetti, R. (2023). Purecake: A verified compiler for a lazy functional language. *Proc. ACM Program. Lang.*, 7(PLDI):952–976.

Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2010). sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115.

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G. (2014). Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA. ACM.

Kumar, R., Myreen, M., Norrish, M., and Owens, S. (2014). CakeML: A verified implementation of ML. In Peter Sewell, editor, *POPL*, pages 179–191, San Diego. ACM.

Kuz, I., Liu, Y., Gorton, I., and Heiser, G. (2007). CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699.

Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J. R., Padon, O., and Parno, B. (2024). Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454, New York, NY, USA. Association for Computing Machinery.

Leroy, X. (2009). Formal verification of a realistic compiler. *CACM*, 52(7):107–115.

Liedtke, J. (1995). On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250.

Liu, Z., Feng, Y., Ni, Y., Li, S., Yin, X., Shi, Q., Xu, B., and Su, Z. (2025). An empirical study of rust-specific bugs in the rustc compiler.

Mi, Z., Li, D., Yang, Z., Wang, X., and Chen, H. (2019). SkyBridge: Fast and secure inter-process communication for microkernels. In *EuroSys*, Dresden, DE. ACM.

Müller, P., Schwerhoff, M., and Summers, A. J. (2016). Viper: A verification infrastructure for permission-based reasoning. In *Int. Conf. Verification, Model Checking & Abstract Interpretation*, pages 41–62, St. Petersburg, FL, US. Springer.

Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G., and Burtsev, A. (2020). RedLeaf: Isolation and communication in a safe operating system. In *USENIX Symposium on Operating Systems Design and Implementation*.

National Institute of Standards and Technology (2017). Bolstering government cybersecurity: Lessons learned from wannacry.

Necula, G. C. (1997). Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA. Association for Computing Machinery.

Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. (2019). Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symposium on Operating Systems Principles*, pages 225–242.

Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. (2017). Hyperkernel: Push-button verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 252–269. ACM.

Neumann, P. G., Boyer, R. S., Bressler, W. J., Boulware, D. W., Levitt, K. N., Feiertag, R. J., Robinson, L., and Heck, M. (1975). A provably secure operating system. Technical Report ESD-TR-75-372, SRI International, Computer Science Laboratory. `https://csrc.nist.rip/publications/history/neum75.pdf`.

Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., and Robinson, L. (1980). A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, Computer Science Laboratory.

Neumann, P. G. and Feiertag, R. J. (2003). PSOS Revisited . In *Computer Security Applications Conference, Annual*, page 208, Los Alamitos, CA, USA. IEEE Computer Society.

Nipkow, T., Paulson, L., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

NXP Semiconductors (2021). i.mx 8m dual / 8m quadlite / 8m quad applications processor data sheet. Quad Cortex-A53 cores up to 1.5 GHz.

O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T., Nagashima, Y., Sewell, T., and Klein, G. (2016). Refinement through restraint: Bringing down the cost of verification. In *ICFP*, Nara, Japan.

Ojeda, M. (2021). Rust for Linux. `https://rust-for-linux.com/`.

Paturel, M., Subasinghe, I., and Heiser, G. (2023). First steps in verifying the seL4 Core Platform. In *Asia-Pacific Workshop on Systems (APSys)*, Seoul, KR. ACM.

Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *TACAS*, pages 151–166, Lisbon, Portugal. Springer.

Pohjola, J. Å., Syeda, H. T., Tanaka, M., Winter, K., Sau, T. W., Nott, B., Ung, T. T., McLaughlin, C., Seassau, R., Myreen, M. O., Norrish, M., and Heiser, G. (2023). Pancake: Verified systems programming made sweeter. In *PLOS*, Koblenz, DE.

Qu, N. (2024). seL4 in software-defined vehicles: Vision, roadmap, and impact at NIO. Keynote at the 6th seL4 Summit.

Rushby, J. (1997). Formal methods and their role in the certification of critical systems. In Shaw, R., editor, *Safety and Reliability of Software Based Systems*, pages 1–42, London. Springer London. Also discusses DO-178B standards for certification in aviation software systems.

seL4 Foundation (2023). seL4 microkit GitHub.

seL4 Foundation (2025). *CAmkES: Component Architecture for microkernel-based Embedded Systems*. seL4 Foundation.

Sewell, T. A. L., Myreen, M. O., and Klein, G. (2013). Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 471–482, New York, NY, USA. Association for Computing Machinery.

Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. (2016). Push-button verification of file systems via crash refinement. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Savannah, GA, US.

Slind, K., Hardin, D., Åman Pohjola, J., and Sproul, M. (2020). Synthesis of verified architectural components for autonomy hosted on a verified microkernel. In *HICSS*, pages 6365–6374, Grand Wilea, Hawaii. ScholarSpace / AIS Electronic Library.

Soller, J. (2015). Redox OS. `https://www.redox-os.org/`.

stack.watch (2024). 2024 security vulnerability statistics.

Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., and Zinzindohoue, J.-K. (2016). Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM.

Tan, Y. K., Heule, M. J. H., and Myreen, M. O. (2021). `cake_lpr`: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*, page 223–241, Berlin, Heidelberg. Springer-Verlag.

Tan, Y. K., Myreen, M., Kumar, R., Fox, A., Owens, S., and Norrish, M. (2019). The verified CakeML compiler backend. *J. Funct. Progr.*, 29.

Thompson, K. (1984). Reflections on trusting trust. *Commun. ACM*, 27(8):761–763.

Trustworthy Systems (2025a). CAmkES: Component architecture for microkernel-based embedded systems. `https://trustworthy.systems/projects/OLD/camkes`.

Trustworthy Systems (2025b). The sel4 microkit. `https://trustworthy.systems/projects/microkit/`.

Tuch, H., Klein, G., and Norrish, M. (2007). Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editor, *POPL*, pages 97–108, Nice, France. ACM.

Wienand, I. and Macpherson, L. (2004). ipbench: A framework for distributed network benchmarking. In *AUUG*, pages 163–170, Melbourne, Australia.

Wortman, D. B., Holt, R. C., Cordy, J. R., Crowe, D. R., and Griggs, I. H. (1981). Euclid: a language for compiling quality software. In *Proceedings of the May 4-7, 1981, National Computer Conference*, AFIPS '81, page 257–263, New York, NY, USA. Association for Computing Machinery.

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in C compilers. In Hall, M. W. and Padua, D. A., editors, *PLDI*, pages 283–294, San Jose, CA, USA. ACM.

Zaostrovnykh, A., Pirelli, S., Iyer, R., Rizzo, M., Argyraki, L. P. K., and Candea, G. (2019). Verifying software network functions with no verification expertise. In *ACM Symposium on Operating Systems Principles*.

Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. (2017). A formally verified NAT. In *ACM Conference on Communications*.

Zhao, J., Legnani, A., Ung, T. T., Truong, H., Sau, T. W., Tanaka, M., Pohjola, J. Å., Sewell, T., Sison, R., Syeda, H., Myreen, M., Norrish, M., and Heiser, G. (2025). Verifying device drivers with Pancake. *arXiv preprint*.