Pancake

Verified Systems Programming Made Sweeter

Johannes Åman Pohjola
j.amanpohjola@unsw.edu.au
UNSW Sydney
Australia

Hira Taqdees Syeda*
syedahir@amazon.com
Chalmers University of Technology
Gothenburg, Sweden

Miki Tanaka
miki.tanaka@unsw.edu.au
UNSW Sydney
Australia

Krishnan Winter
k.winter@student.unsw.edu.au
UNSW Sydney
Australia

Tsun Wang Sau
t.sau@student.unsw.edu.au
UNSW Sydney
Australia

Benjamin Nott
b.nott@student.unsw.edu.au
UNSW Sydney
Australia

Tiana Tsang Ung
t.tsangung@student.unsw.edu.au
UNSW Sydney
Australia

Craig McLaughlin
c.mclaughlin@unsw.edu.au
UNSW Sydney
Australia

Remy Seassau†
remy.seassau@cs.ox.ac.uk
UNSW Sydney
Australia

Magnus O. Myreen
myreen@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Michael Norrish
michael.norrish@anu.edu.au
Australian National University
Canberra, Australia

Gernot Heiser
gernot@unsw.edu.au
UNSW Sydney
Australia

Abstract

We introduce Pancake, a new language for verifiable, low-level systems programming, especially device drivers. Pancake eschews complex type systems to make the language attractive to systems programmers, while at the same time aiming to ease the formal verification of code. We describe the design of the language and its verified compiler, and examine its usability, performance and current limitations through case studies of device drivers and related systems components for an seL4-based operating system.

ACM Reference Format:
Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. 2023. Pancake: Verified Systems Programming Made Sweeter. In *12th Workshop on Programming Languages and Operating Systems (PLOS '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3623759.3624544

*Now with Amazon Web Services.
†Now at University of Oxford.

PLOS '23, October 23, 2023, Koblenz, Germany

1 Introduction

Low-level systems programming is notoriously error-prone. While widespread use of the C programming language is not the only culprit, its unsafety and complicated semantics certainly do not help. A significant amount of programming language research has therefore been invested in the creating programming languages for safe systems programming.

In this paper, we introduce a new systems programming language, *Pancake*, aimed at enabling formal verification of real-world device drivers. Unlike previous languages, the design of Pancake emphasises ease of verification over achieving safety via, e.g., type safety or language restrictions.

Why device drivers? Driver bugs are the leading cause of OS compromise, accounting for the majority of the 1,057 CVEs reported for Linux in the period 2018–22 [MITRE Corporation 2023]—clearly they should be the #1 targets of OS verification efforts.

Furthermore, when starting with the formally verified seL4 microkernel [Klein et al. 2014], all the remaining OS code that directly interfaces to hardware is in device drivers. Hardware interfaces require dealing with hardware-specified data layouts and access protocols, something not possible in most high-level languages. Making a language suitable for drivers ensures that it can be used for all OS code.

Why not C? While C is the de-facto standard systems language, from a verification standpoint, C's semantics has a number of undesirable properties: a complicated memory model, underspecified order of evaluation, and the need

to prove the absence of undefined behaviour at almost every step. While the seL4 verification demonstrated that these challenges can be overcome, even when verifying machine code without relying on formal properties of a compiler [Sewell et al. 2013], the cost was high: $350/SLOC just for verifying the C code, and this cost continues to impact evolution of the kernel. While using a verified compiler [Leroy 2009] with the verification toolchain VST [Appel 2011] can help, this has to date not resulted in verified real-world drivers.

Why not type safety? A popular approach to better systems programming languages is incorporating advanced language features that make certain safety properties hold by construction. For example, Cogent has a linear type discipline that prevents memory leaks [Amani et al. 2016], Rust's borrow checker enforces ownership and lifetimes [Klabnik and Nichols 2017], and Cyclone incorporates garbage collection and ML-style polymorphism [Jim et al. 2002].

Such features can eliminate whole classes of bugs, or at least reduce bug density, at the cost of making the language semantics and implementation more complicated. Yet they still fall short of ensuring full functional correctness, which is the main focus of our work, and it is unclear how helpful advanced language features are in achieving this goal.

What then? Our working hypothesis is that in order to drive down the cost of verified systems code, we need a language *less complicated* than C, but not necessarily a *safer* language. Consider the following points:

- Functional correctness proofs produce (and consume) significantly stronger properties than type systems typically guarantee. Imagine proving this Hoare triple:

  $$\{n \in \mathbb{N}\}\ x := \mathsf{fib}(n)\ \{x = n^{\text{th}}\ \text{Fibonacci no.}\}$$

  A sound static type checker might give us the following information nuggets:

  $$\mathsf{fib} :: \mathsf{int} \to \mathsf{int} \quad n :: \mathsf{int} \quad x :: \mathsf{int},$$

  but their value to a correctness proof is unclear. The type annotation on n gives us strictly less information than the precondition, and the type annotation on x gives us strictly less information than the postcondition. Knowing the type of fib does not obviate the need to dig into its implementation details in the proof.
  A stronger type system could give more useful information, but unless it is so powerful (and so undecidable) as to be a full-featured proof calculus, the information we need will almost always be stronger than what the type system provides.
- The information provided by a type system is only useful if the type system is sound. Most practical languages have unverified type systems, or type systems with known soundness bugs. Type systems can be verified [Naraschewski and Nipkow 1999], but type soundness proofs tend to be delicate, and have subtle interactions with seemingly minor changes to a language. Maintaining a type soundness proof for a living language can significantly bog down development.
- The safety guarantees of a language only hold if no backdoors are used. However, in low-level systems programming, it is often necessary to break out of a type-safe environment. For example, device driver code must adhere to hardware-specified data locations, layouts and access protocols. Hence driver code written in safe languages must use significant amounts of unsafe code, effectively escapes to C [Astrauskas et al. 2020; Evans et al. 2020], which mostly eliminates the benefit of using a safe language.
- A simple formal semantics will be amenable to simpler proofs, while more safety features tend to make the semantics more complicated. Provided such a formal semantics exists in the first place: despite years of research [Jung et al. 2018; Kan et al. 2018; Wang et al. 2018; Weiss et al. 2019], there is still no complete formal specification of Rust.

Enter Pancake. We aim for a radically minimal design that offers a sufficiently expressive interface for writing low-level systems programs, such as device drivers, alongside a number of advantages for formal verification. Most importantly, the language is completely specified by a straightforward formal semantics that fits in a few hundred lines of HOL4 code, with a simple memory model, no notion of undefined behaviour, and no ambiguities in evaluation order.

Pancake is an unmanaged language with no static type system, at a level of abstraction between C and assembly. The type system and memory model are kept as simple as possible: the only kinds of data are machine words, code pointers, and structs. Programs cannot inspect the stack, which simplifies semantics. The heap is statically allocated; there is no equivalent of `malloc` and `free`. There are no concurrency primitives—making drivers single-threaded [Ryzhyk et al. 2009, 2010] significantly simplifies verification, maps well onto the modular design of microkernel-based OSes, and is routinely used for drivers on seL4 without undue impact on performance [Heiser et al. 2022].

Contributions. Pancake is work in progress; we have not yet completed a device-driver verification case study, hence cannot yet confirm our working hypothesis. Nevertheless, we present two key technical contributions:

1. A formally verified end-to-end compiler (Section 2), which allows safety and liveness properties of Pancake programs to carry over to the machine code that runs them. Our compiler reuses significant parts of the compiler backend and proofs for the CakeML compiler [Tan et al. 2019]. Crucially, to preserve liveness properties, the compiler

performs a sound static analysis that guarantees the absence of premature termination arising from stack overflows. The compiler and its verification is free software, and available online.[1]

2. We demonstrate that Pancake is usable by systems programmers brought up on C, as confirmed by a case study (Section 3) consisting of serial drivers and multiplexer components for serial and network devices for the seL4 Core Platform [Heiser et al. 2022]. While the study shows a performance penalty compared to equivalent components written in C, our networking system still manages to outperform Linux.

Below we discuss the design of Pancake and the two contributions, as well as on-going and planned work (Section 4).

2 Pancake Overview

Pancake leverages the compiler backend of CakeML [Kumar et al. 2014], an impure functional programming language similar to Standard ML, with an optimising compiler verified all the way from source syntax to machine code. This eliminates the compiler from the Trusted Computing Base (TCB), and avoids the need for fragile validation of the compiler output on a program-by-program basis [Sewell et al. 2013].

CakeML itself is not suited for low-level systems programming in resource-constrained environments, where predictable performance is important. Its memory management is all handled by the language runtime, and memory allocation may trigger a stop-the-world garbage collector.

Pancake, in contrast, is explicitly designed to be close to hardware, and not managed. Yet, by integration into the CakeML ecosystem, it can reuse many of the existing correctness proofs for the CakeML compiler,[2] and possibly other infrastructure developed for verification of CakeML code.

2.1 Design

For ease of verification and predictability of compilation, we keep Pancake's type system and memory model as simple as possible. The type system has only three kinds of data:

1. *Machine words* of the architecture's word size.
2. *Labels*, which are essentially code pointers.
3. *Structs*, whose fields are machine words, labels, or nested structs.

Local variables are stack-allocated, and the language does not allow pointers into the stack. Data may also be allocated on a static heap: there is no equivalent of `malloc` and `free`. Reads and writes to the heap can be performed byte-wise or word-wise.

[1] http://code.cakeml.org for source, or http://cakeml.org for pre-packaged versions. Note that Pancake is fully integrated into the CakeML compiler, rather than a stand-alone release.

[2] Specifically, we can reuse the correctness proofs of everything from WordLang downwards in Figure 3, and the compiler bootstrapping machinery.

```
exp  :=  Const word  |  Var string  |  Label string
     |   Struct exp⋆  |  Field num exp
     |   Load shape exp  |  LoadByte exp
     |   Op binop exp⋆  |  Cmp cmp exp exp
     |   Shift shift exp num  |  BaseAddr

prog :=  Skip  |  Dec string exp prog
     |   Assign string exp  |  Store exp exp
     |   StoreByte exp exp  |  Seq prog prog
     |   If exp prog prog  |  While exp prog
     |   Break  |  Continue  |  Call ret exp exp⋆
     |   Raise string exp  |  Return exp  |  Tick
     |   ExtCall string exp exp exp exp
```

Figure 1. Abstract syntax of Pancake.

```
while true {
   #tx_fifo_busy(tmp_c_uart, tmp_clen_uart,
                 tmp_a_uart, tmp_alen_uart);
   tx_fifo_ret = ldb tmp_a_uart;
   if tx_fifo_ret <> 1 {
     strb c_arr_uart, tmp;
     #putchar_regs(c_arr_uart, clen_uart,
                   a_arr_uart, alen_uart);
     break;
   }
}
```

Figure 2. Pancake code snippet (concrete syntax)

Figure 1 shows the current abstract syntax of Pancake, divided into expressions (*exp*) and statements (*prog*). It includes the standard structured programming primitives, operators for manipulating words, and primitives for reading from and writing to the heap. The programmer is responsible for declaring whether function calls (Call) are tail calls or returning calls by specifying *ret* appropriately. *ret* may also contain exception handlers, which will catch any exceptions raised (using Raise) by the callee. ExtCall represents foreign function calls, and is currently the only way for Pancake programs to interact with the outside world. Data is passed using byte arrays, allocated from within Pancake's statically allocated heap.

Expressions are designed to be completely free of side effects: function calls are statements, and that there is no equivalent of the ++ operators familiar from e.g. C. This simplifies verification and compilation by making the evaluation order of expressions immaterial.

The flavour of the concrete syntax is illustrated by the code snippet in Figure 2. It is taken from the serial driver discussed in Section 3. Foreign function calls (prefixed with #) are used to communicate with the device.

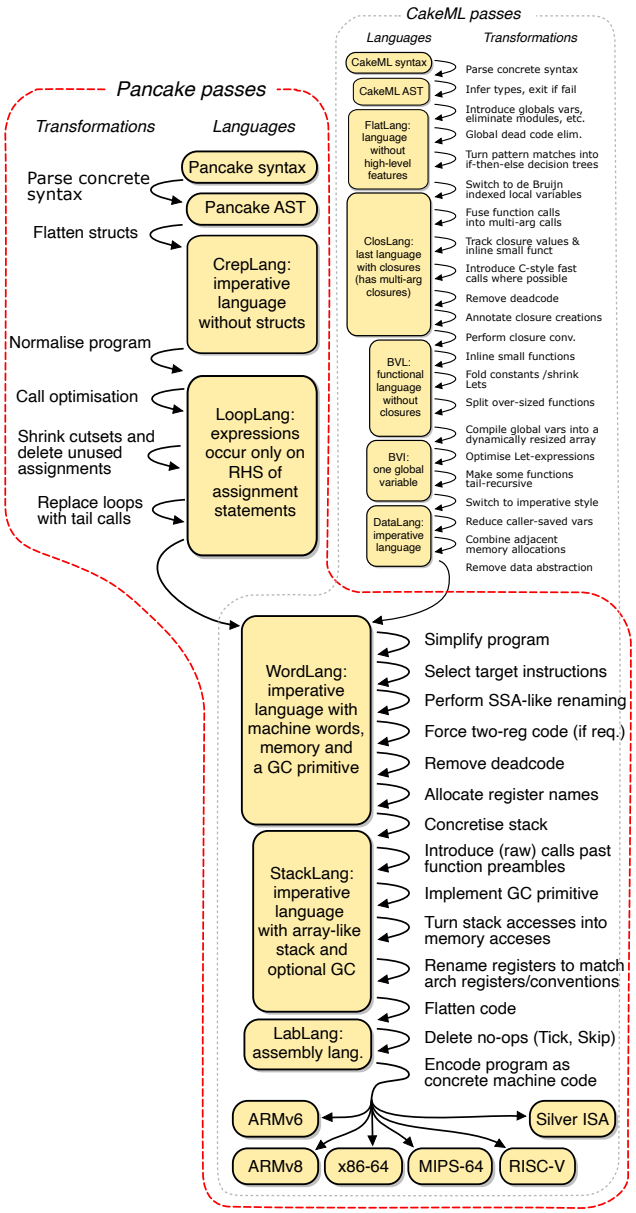

Figure 3. Overview of CakeML and Pancake compiler stack.

2.2 Compiler

Figure 3 illustrates how the Pancake compiler fits into the CakeML compiler backend. Each box denotes an intermediate language, and each arrow denotes a compiler pass.

The Pancake compiler enters the CakeML compiler backend in the WordLang intermediate language. WordLang is assembly-like in its very low-level data representation (machine words are the only type), but in terms of control flow it is a structured programming language similar to C. Some features of WordLang, such as its garbage collection primitive, are useful when compiling functional programs, but not used by Pancake.

The Pancake compiler goes through two intermediate languages before WordLang. The first, CrepLang, is very similar to Pancake but does not feature structs. Thus, the main purpose of this compiler pass is to flatten structs. The second, LoopLang, allows expressions only on the RHS of assignment statements (c.f. Figure 1), and annotates loops with liveness information for its stack variables. This facilitates the eventual transition from structured code to instruction sequences, and allows us to compile away loops using tail recursion (because loops are absent in WordLang).

The compiler is implemented as shallowly embedded functions in the HOL4 theorem prover [Slind and Norrish 2008], as well as a stand-alone binary compiler obtained by in-logic bootstrapping [Myreen 2021]. The former is used for in-logic evaluation, which can generate very strong certificate theorems about the machine code produced by individual compiler runs. The latter is suitable for a more humane workflow, with compilation times in seconds instead of hours.

2.3 Semantics

Pancake inherits CakeML's semantics style, *functional big-step semantics* [Owens et al. 2016], which uses an evaluation function from programs to results. In standard relational big-step semantics, a program is given meaning by a *relation* between programs and results. The functional style is similar to a language interpreter, but not necessarily executable.

All the intermediate languages shown as coloured boxes in Figure 3 have this kind of semantics. This style simplifies formal proofs of compiler correctness by making the semantics more amenable to term rewriting, at the expense of making the treatment of non-termination awkward. The top-level semantics of a Pancake program is an element of the following datatype:

```
datatype behaviour = Diverge (io_event llist)
                   | Terminate outcome (io_event list)
                   | Fail
```

The key point is that a program's *behaviour* is defined in terms of a finite (*list*) or possibly infinite (*llist*) trace of I/O events. An *io_event* denotes a returning foreign function interface (FFI) call (ExtCall from Figure 1), together with its arguments and return value.

Pancake programs that communicate with the outside world via the FFI have semantics that is parameterised on a *foreign function oracle* modelling the outside world. The oracle describes the effects of FFI calls: how they change the state of the outside world, and what data they pass back to the Pancake program.

This model is sufficiently expressive to enable verification of several key correctness properties. First, we can prove the absence of crashes and undefined behaviour arising from generic programming faults, by proving that drivers cannot

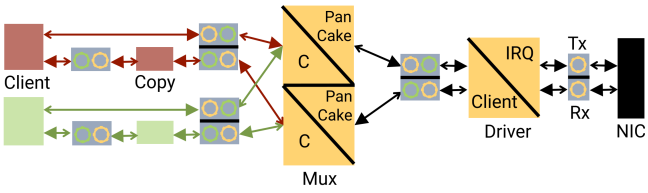

Figure 4. sDDF networking architecture. Coloured boxes are modules, grey boxes shared data structures.

Fail. More interestingly, we can show compliance with protocols for interacting with devices and operating systems, by proving temporal properties about the *io_event*s. Real-time properties are often important in device drivers, but are for now left out of scope; but as a first approximation, we can include events that model delays in the *io_event*s without changing the current model.

2.4 Compiler Correctness

We have verified the Pancake compiler backend in HOL4. That is, we proved that for any Pancake program that does not Fail, the *behaviour* of the Pancake source program and the generated machine code is equal.

The verification proves the absence of out-of-memory errors: we have implemented and verified a compilation phase that can statically calculate an upper bound on stack usage, and proved this bound correct using our previous work on space-cost semantics [Gómez-Londoño et al. 2020]. This tells systems designers exactly how much space they must reserve for the Pancake stack in a statically allocated environment. Obviously, this does not apply to programs featuring non-tail recursion, which can exhaust the stack, but such recursion is generally not useful for systems code such as drivers.

We proved the correctness of Pancake-specific compiler phases (Pancake to WordLang), and then composed with the correctness proof from WordLang down to the target established from the CakeML backend proof. This composition required some care in minimising the implication of CakeML language features (e.g. garbage collection) as well as in setting out low-level information that Pancake needs.

3 Case Study: Serial Driver and Multiplexers

Figure 4 shows a modularised networking system based on the seL4 Device Driver Framework (sDDF) [Parker 2023], which is designed to provide high-performance I/O for an seL4-based high-assurance operating system (OS) using the seL4 Microkit [Heiser et al. 2022; Trustworthy Systems 2023]. The design emphasises implementation simplicity for robustness and verification, with each component a sequential, event-driven program in its own address space. Components use shared-memory communication synchronised by seL4-provided semaphores.

The design emphasises strong separation of concerns: each component has a single purpose, e.g. the device driver solely translates a hardware-specific device interface to a hardware-independent device-class interface. Devices are shared between multiple clients using explicit multiplexer (Mux) components, and explicit copiers are used where required to isolate clients from each other.

Currently, each Pancake sDDF component consists of at least two files, one C and one Pancake. The C component contains a set of functions callable via Pancake's FFI (Section 2.3). The Pancake component contains the majority of the logic, and utilises the FFI for interacting with memory-mapped device registers, and the shared memory regions used for inter-component communication. The C code also allocates the memory used for Pancake's stack and heap.

We implement three components in Pancake:

Serial Driver: existing driver transcribed from C;
Serial Mux: new multiplexer for serial driver;
Ethernet Mux: existing receive (Rx) and transmit (Tx) Ethernet Muxes, plus Rx Copier, transcribed from C.

Table 1 compares the source-code sizes of the Pancake (+C) Ethernet Mux and the original C-only implementation. A portion of the Pancake C code is taken from the `basis_ffi` template (from the CakeML repository), which includes some helper functions for converting variables to byte sized arrays, as well as initialising Pancake's memory regions. The above figures also exclude the boilerplate `cml_exit()` and `cml_clear()` functions, which originally are used to destroy the process after Pancake has finished execution, and clear Pancake's code buffer region from the instruction cache. These functions are expected by the Pancake compiler but as we do not invoke these functions, they can be left stubbed out in our implementation. A significant contributor to the Pancake C total is the packing and unpacking of arguments and return values.

We evaluate the performance of the Pancake Muxes against the C versions by interfacing them to the same C Ethernet driver, and a single client, which receives UDP packets and sends them back unmodified. We evaluate this configuration on a Freescale i.MX 8M Mini quad SoC with 2 GiB RAM running at 1.2 GHz, restricted to using a single core. Our system uses a minimally adapted lwIP protocol stack [Dunkels 2001], which is linked against its client. We use ipbench [Wienand and Macpherson 2004], running on

Component	Pan Code	Pan C Code	Pan Total	C Total
mux_tx	81	206	287	85
mux_rx	179	314	493	222

Table 1. Comparison of source code size (SLOC) of Pancake Mux components and the original C implementation.

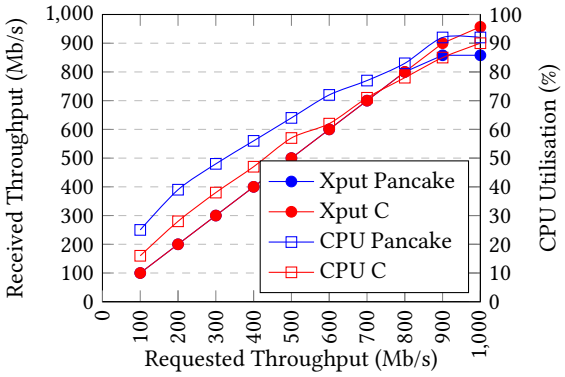

Figure 5. Performance of Ethernet multiplexer written in Pancake vs C, in terms of achieved throughput (Xput) and observed CPU utilisation (CPU).

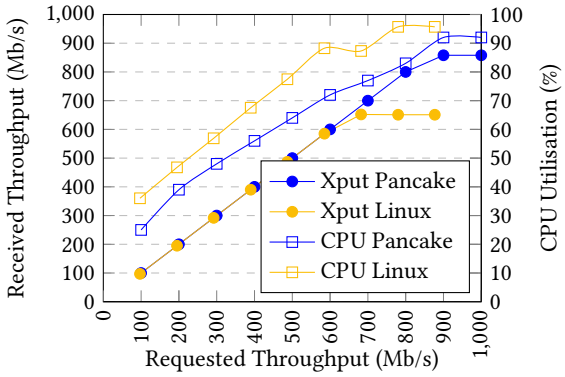

Figure 6. Linux vs Pancake Mux Benchmark

a 10-node x86 cluster to generate network load, measuring achieved throughput and latency of the returned packets.

Figure 5 shows the received throughput and CPU utilisation of the two implementations. We see that the Pancake implementation handles the load until we reach requested throughputs of 900 Mb/s, where CPU utilisation reaches 92% and throughput stagnates at 858 Mb/s, whereas the C implementation achieves wire speed. At 100 Mb/s throughput (where no batching occurs), CPU load of C is 16%, compared to 25% for Pancake, a 50% overhead.

One cause of this overhead is the re-running of initialisation code when entering Pancake. Pancake presently expects to be a stand-alone program that starts and runs to completion. In contrast, we invoke Pancake as an event handler, requiring redundant execution of initialisation code. This initialisation code is auto-generated by the compiler and placed at the beginning of the `main` function in the assembly output as a series of `.byte` instructions. When compiling a Pancake program consisting of a single return statement, the compiler generates an assembly file with 211 SLOC, with 103 of these lines being the `.byte` instructions in the `main` function. We plan to remove the need for this in future work.

Another source of overhead is that the Pancake compiler optimises less aggressively than C. The reference C implementation is compiled with the highest level of optimistations available. However, in our Pancake implementation, the Pancake segments of the Rx and Tx Mux components do not receive such optimisations. When building the reference C implementation with no optimisations enabled for the Rx and Tx components, we find similar trends to the original Pancake benchmarks. We achieve a maximum of 828 Mb/s at 1 Gb/s requested throughputs, with CPU utilisation peaking at 94%.

Figure 6 compares the Pancake/seL4 setup to a Linux v6.1.1 system using `buildroot` and running on the same hardware, with a simple user program which reads from a socket using the `recvmsg()` system call, and then returns the packets using `sendmsg()`. We see that Linux throughput plateaus at 700 Mb/s, when it saturates the CPU. Linux CPU load at 100 Mb/s is 36%, 44% above that of the Pancake configuration.

4 Current and Future Work

While still a work in progress, Pancake is already usable for low-level systems programming (Section 3), with strengthened assurance resulting from a verified compiler (Section 2). Here we outline some of our current and planned activities for making the language fully fit for purpose and to build a verification framework for Pancake programs.

4.1 Shared-Memory Semantics

Currently, the Pancake semantics assumes exclusive ownership of the statically allocated heap memory region and that this memory is not observable by the environment. These assumptions do not hold for memory-mapped device registers, which are special memory locations used for interfacing with device hardware. Nor does it hold when devices directly write to memory (DMA) which is also accessed by the driver.

This limitation forces awkward workarounds (Section 3) and code bloat (Table 1): Any interaction with the device is mediated by FFI calls to C, adding a layer of indirection as part of the TCB. Native Pancake support for interacting with shared memory will eliminate this indirection.

The key research challenge is that the ISA models used by the compiler backend [Tan et al. 2019] are inherently sequential, and 100+ KSLOC of compiler correctness proofs inextricably rely on this fact. Previous work has integrated driver models for specific devices directly into an ARM ISA model with an interleaving semantics [Alkassar et al. 2007]. This approach would have to abandon much of the existing proof base, and would not provide the modularity we need for targeting multiple ISAs and devices. We are therefore parameterising the language semantics on a model for shared memory, which supports proof reuse and provides flexibility for incorporating arbitrary devices.

We have introduced such a shared memory semantics into the lower part of the compiler and verified the compilation phase from assembly (LabLang in Figure 3) to machine code.

4.2 Interaction Tree Semantics

As discussed in Section 2.3, Pancake programs that communicate with the outside world have their semantics parameterised on an *oracle* that describes how the outside world responds to observable events: how the world changes and what data it makes available to the Pancake program. The oracles are deterministic, which simplifies definitions and compiler proofs but prevents modelling realistic devices, which have inherent non-determinism in their interactions with the outside world. The oracle model also requires device models to be provided up-front, making it difficult to decouple reasoning about code and about devices.

A more promising approach is interaction trees [Xia et al. 2020], which represent the behaviour of a program as a (possibly infinite) coinductive tree, with nodes representing actions and branches representing possible environmental responses. This model naturally accommodates non-determinism, and obviates the need to have an oracle at all.

We have developed an interaction tree semantics for Pancake. We are currently proving it sound and complete, and are in the process of using it to verify the device drivers described in Section 3.

4.3 Decompilation Framework

Pancake is designed to give programmers the fine-grained control of low-level details necessary for systems programming. But for verification, reasoning at a higher level of abstraction is crucial for scalability. We plan to leverage previous work on *decompilation into logic* [Myreen et al. 2008], where higher-level models are automatically generated from the program source code, together with proofs that the model accurately describes the behaviour of the program, enabling high-level proofs about low-level code with no gap in the trust story.

While decompilation has been used on C [Greenaway et al. 2014] and machine code [Myreen et al. 2008], this earlier work verified non-interactive code using traditional pre- and post-conditions, techniques that are applicable only to terminating programs. In contrast, device drivers are interactive and should never terminate, requiring decompilation techniques that target event-based specifications of potentially non-terminating programs.

5 Related Work

We aim to use Pancake to make driver verification scale to real-world, non-trivial drivers without cutting corners on realism or soundness. Previous attempts to verify drivers either left significant gaps between the analysed model and the real code [Kim et al. 2008; Möre 2021; Penninckx et al. 2012], or failed to demonstrate scalability beyond the most simple serial drivers [Alkassar et al. 2007; Alkassar and Hillebrand 2008; Chen et al. 2016; Duan and Regehr 2010]. Specifically such work did not deal with core characteristics of real drivers, such as direct memory access (DMA) by the device. Pancake will address these issues and our case study will be a real-world ethernet driver.

We aim to prove drivers correct at the level of a convenient source language, and then transport the proved properties to the level of the machine code that actually runs. The Verified Software Toolchain [Appel 2011] achieves similar aims using a retargettable Separation Logic [Appel et al. 2014] on top of CompCert C and the verified CompCert compiler [Leroy 2009]. However, when using VST, users developing verification proofs are forced to deal with the intricacies that arise from a C semantics. This can be mitigated if the programmer takes care to use only a well-behaved subset of C. But even then, even very basic operations such as integer arithmetic can potentially trigger undefined behaviour, and this must be considered in rigorous proofs. Pancake invites us to ask: would it not be easier if this well-behaved subset was just the whole language? Finally, we note that to the best of our knowledge, VST has not been used for verification of device drivers.

The earlier Cogent project [Amani et al. 2016] had much the same aims as Pancake. It stalled however, partly because systems programmers found it difficult to understand the language, and work around its restrictions.

6 Conclusion

We have introduced Pancake, a programming language designed for verifiable low-level systems programming. Our initial experience with Pancake is encouraging, despite more work being needed to create a language and verification framework that are fully fit for purpose.

First, our case studies show that systems programmers used to C find the language easy to understand and adapt to, and sufficiently expressive to develop reasonably performant device drivers. Second, the decision to build atop CakeML has paid off from a proof-engineering point of view: we have been able to reuse proofs for several key compilation passes, including in particular register allocation.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work was partially supported by the Technology Innovation Institute (TII), by the UK National Cyber Security Centre (NCSC), and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev. 2007. Formal Device and Programming Model for a Serial Interface. In *International Verification Workshop*. Bremen, DE, 4–20.

Eyad Alkassar and Mark A. Hillebrand. 2008. Formal Functional Verification of Device Drivers. In *Verified Software: Theories, Tools and Experiments (Lecture Notes in Computer Science, Vol. 5295)*. Springer, Toronto, Canada, 225–239.

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188. https://doi.org/10.1145/2872362.2872404

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *European Symposium on Programming (ESOP) (LNCS, Vol. 6602)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press, USA.

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe Rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 136:1–136:27. https://doi.org/10.1145/3428204

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 431–447. https://doi.org/10.1145/2908080.2908101

Jianjun Duan and John Regehr. 2010. Correctness Proofs for Device Drivers in Embedded Systems. In *Systems Software Verification*. USENIX Association, Vancouver, BC, CA.

Adam Dunkels. 2001. *Minimal TCP/IP implementation with proxy support*. Technical Report T2001-20. SICS. 81 pages. http://www.sics.se/~adam/thesis.pdf.

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *International Conference on Software Engineering*. 246–257. https://doi.org/10.1145/3377811.3380413

Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 204:1–204:29. https://doi.org/10.1145/3428272

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh, UK, 429–439. https://doi.org/10.1145/2594291.2594296

Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. 2022. Can We Put the "S" Into IoT?. In *IEEE World Forum on Internet of Things*. Yokohama, JP.

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*. USENIX, 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 2018. K-Rust: An Executable Formal Semantics for Rust. *CoRR* abs/1804.07608 (2018). http://arxiv.org/abs/1804.07608 Preprint.

Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. 2008. Formal Verification of a Flash Memory Device Driver – An Experience Report. In *SPIN Workshop on Model Checking Software (Lecture Notes in Computer Science, Vol. 5156)*. Los Angeles, CA, US, 144–159. https://doi.org/10.1007/978-3-540-85114-1_12

Steve Klabnik and Carol Nichols. 2017. *The Rust Programming Language*. No Starch Press.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. https://doi.org/10.1145/2560537

Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, San Diego, 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

MITRE Corporation. 2023. Linux ≫ Linux Kernel: Security Vulnerabilities (CVSS score ≥ 9). https://www.cvedetails.com/vulnerability-list.php?vendor_id=33&product_id=47&version_id=&page=1&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=9&cvssscoremax=0&year=0&month=0&cweid=0&order=1&trc=3034&sha=544260ec3a86a7e17f8b02b39d6342815d8d4bd5 Accessed: 2023-01-25.

Tomas Möre. 2021. *Formal verification of device driver monitors in HOL 4*. Masters Thesis. School of EECS, KTH, SE.

Magnus O. Myreen. 2021. A Minimalistic Verified Bootstrapped Compiler (Proof Pearl). In *Certified Programs and Proofs (CPP)*. ACM, 32–45. https://doi.org/10.1145/3437992.3439915

Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2008. Machine-code Verification for Multiple Architectures: An Application of Decompilation into Logic. In *Proceedings of the 2008 Conference on Formal Methods in Computer-Aided Design*. IEEE, Portland, OR, US.

Wolfgang Naraschewski and Tobias Nipkow. 1999. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.* 23, 3-4 (1999), 299–318. https://doi.org/10.1023/A:1006277616879

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP) (LNCS, Vol. 9632)*. Springer, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Lucy Parker. 2023. *The seL4 Device Driver Framework*. https://trustworthy.systems/publications/papers/Parker_23%3Asel4s.abstract Talk at the 5th seL4 Summit.

Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Sound Formal Verification of Linux's USB BP Keyboard Driver. In *NASA Formal Methods Symposium (Lecture Notes in Computer Science, Vol. 7226)*. https://doi.org/10.1007/978-3-642-28891-3_21

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009. Dingo: Taming Device Drivers. In *EuroSys Conference*. Nuremberg, DE, 275–288.

Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. 2010. The Case for Active Device Drivers. In *Asia-Pacific Workshop on Systems (APSys)*. New Delhi, India, 25–30.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Seattle, Washington, USA, 471–481.

Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs) (LNCS, Vol. 5170)*. Springer, 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

Yong Kiam Tan, Magnus Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (Feb. 2019), 57 pages. https://doi.org/10.1017/S0956796818000229

Trustworthy Systems. 2023. *The seL4 Microkit*. UNSW Sydney. https://trustworthy.systems/projects/microkit/

Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*. IEEE Computer Society, 44–51. https://doi.org/10.1109/TASE.2018.00014

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). http://arxiv.org/abs/1903.00982 Preprint.

Ian Wienand and Luke Macpherson. 2004. ipbench: A Framework for Distributed Network Benchmarking. In *Conference for Unix, Linux and Open Source Professionals (AUUG)*. Melbourne, Australia, 163–170.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4 (Jan. 2020), 51:1–51:32. https://doi.org/10.1145/3371119