Pancake: Verified Systems Programming Made Sweeter

Johannes Åman Pohjola [1] Hira Taqdees Syeda [2] Miki Tanaka [1] Krishnan Winter [1] Tsun Wang Sau [1] Benjamin Nott [1] Tiana Tsang Ung [1] Craig McLaughlin [1] Remy Seassau [1] Magnus O. Myreen [2] Michael Norrish [3] Gernot Heiser [1]

[1]UNSW Sydney

[2]Chalmers University of Technology [3]Australian National University

PLOS '23

Outline

Introduction

Background

Design

Case Study

Future Work

Q&A

Section 1

Introduction

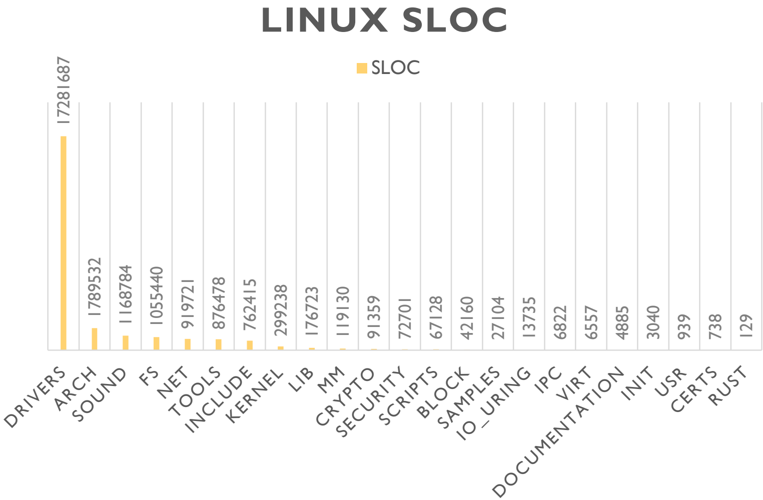
What is Pancake?

Pancake is a new language for low-level systems programming, aiming to promote the ease of formal verification.

Section 2

Background

Why do we need Pancake?



Why not use C?

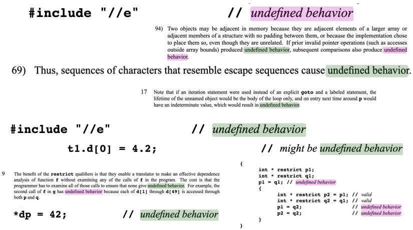
C is the defacto systems programming language, so why not verify C code?

- C has many undesirable properties for verification.

Why not use C?

C is the defacto systems programming language, so why not verify C code?

- C has many undesirable properties for verification.



```
#include "//e"                // undefined behavior
```

94) Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

69) Thus, sequences of characters that resemble escape sequences cause undefined behavior.

17 Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around **p** would have an indeterminate value, which would result in undefined behavior.

```
#include "//e"                // undefined behavior
```

```
t1.d[0] = 4.2;                // might be undefined behavior
```

9 The benefit of the **restrict** qualifiers is that they enable a translator to make an effective dependence analysis of function **f** without examining any of the calls of **f** in the program. The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example, the second call of **f** in **g** has undefined behavior because each of **d[1]** through **d[49]** is accessed through both **p** and **q**.

```
*dp = 42;             // undefined behavior
```

```
{
        int * restrict p1;
        int * restrict q1;
        p1 = q1; // undefined behavior
        {
                int * restrict p2 = p1; // valid
                int * restrict q2 = q1; // valid
                p1 = q2;                // undefined behavior
                p2 = q2;                // undefined behavior
        }
}
```

Why not use C?

C is the defacto systems programming language, so why not verify C code?

- C has many undesirable properties for verification.
- The seL4 verification effort demonstrated it was possible.

10,000 SLOC

$350 Per SLOC

22 Person Years

Why not type safety?

Why not take advantage of type safety? Why not a language such as Rust?

- The addition of these advanced language features increase the complexity of the language.

Why not type safety?

Why not take advantage of type safety? Why not a language such as Rust?

- It falls short of ensuring full functional correctness:
  - Use of unsafe.
  - Unverified compiler.
  - Unverified run-time.
  - No formal semantics.

What do we aim to achieve?

PANCAKE
A Language for Verified Systems Programming

Enter Pancake!

- Minimal design that still remains sufficiently expressive for writing systems code.
- We don't strictly want a safer language,
- But rather a language that is less complicated and more ammenable to verification.

Section 3

Design

Pancake Overview

PANCAKE
A Language for Verified Systems Programming

Pancake is a new ”C-like” systems programming language.

- It is an unmanaged language.
- Simple type system.
- No stack inspection.
- Statically allocated heap.
- No concurrency primitives.

Pancake Overview

PANCAKE
A Language for Verified Systems Programming

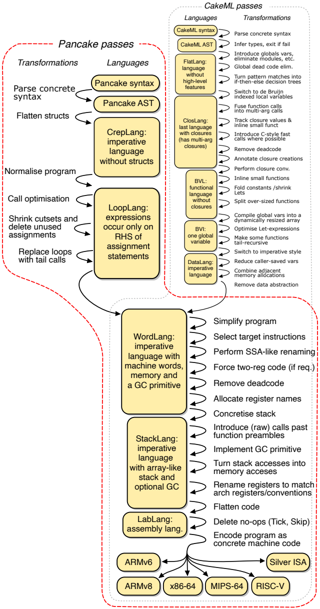
Pancake is a new "C-like" systems programming language.

- It is an unmanaged language.
- Simple type system.

- No stack inspection.
- Statically allocated heap.
- No concurrency primitives.

Compiler

The Pancake compiler is formally verified from end to end!



CakeML passes

Languages

Transformations

CakeML syntax

Parse concrete syntax

CakeML AST

Infer types, exit if fail

Introduce globals vars, eliminate modules, etc.

FlatLang: language without high-level features

Global dead code elim.

Turn pattern matches into if-then-else decision trees

Switch to de Bruijn indexed local variables

ClosLang: last language with closures (has multi-arg closures)

Fuse function calls into multi-arg calls

Track closure values & inline small funct

Introduce C-style fast calls where possible

Remove deadcode

Annotate closure creations

Perform closure conv.

BVL: functional language without closures

Inline small functions

Fold constants /shrink Lets

Split over-sized functions

Compile global vars into a dynamically resized array

BVI: one global variable

Optimise Let-expressions

Make some functions tail-recursive

Switch to imperative style

DataLang: imperative language

Reduce caller-saved vars

Combine adjacent memory allocations

Remove data abstraction

Pancake passes

Transformations

Languages

Pancake syntax

Parse concrete syntax

Pancake AST

Flatten structs

CrepLang: imperative language without structs

Normalise program

Call optimisation

LoopLang: expressions occur only on RHS of assignment statements

Shrink cutsets and delete unused assignments

Replace loops with tail calls

WordLang: imperative language with machine words, memory and a GC primitive

Simplify program

Select target instructions

Perform SSA-like renaming

Force two-reg code (if req.)

Remove deadcode

Allocate register names

Concretise stack

StackLang: imperative language with array-like stack and optional GC

Introduce (raw) calls past function preambles

Implement GC primitive

Turn stack accesses into memory acceses

Rename registers to match arch registers/conventions

Flatten code

LabLang: assembly lang.

Delete no-ops (Tick, Skip)

Encode program as concrete machine code

ARMv6

Silver ISA

ARMv8

x86-64

MIPS-64

RISC-V

Type System

Pancake has a very simple type system, with only 3 kinds of data:

- *Machine Words*
- *Labels*
- *Structs*

Foreign Function Interface (FFI)

Pancake offers a Foreign Function Interface, that allows Pancake code to interact with the outside world.

```
#ffihello_world(a, alen, b, blen);
// Calling a C function named "hello_world"
```

Foreign Function Interface (FFI)

Pancake offers a Foreign Function Interface, that allows Pancake code to interact with the outside world.

```
#ffihello_world(a, alen, b, blen);
// Calling a C function named "hello_world"
```

Foreign Function Interface (FFI)

Pancake offers a Foreign Function Interface, that allows Pancake code to interact with the outside world.

```
#ffihello_world(a, alen, b, blen);
// Calling a C function named "hello_world"
```

```
void ffihello_world(char *a, unsigned int alen,
                    char *b, unsigned int blen)
{
    printf("Hello World");
}
// The C function that we are calling
```

Foreign Function Interface (FFI)

Pancake offers a Foreign Function Interface, that allows Pancake code to interact with the outside world.

```
#ffihello_world(a, alen, b, blen);
// Calling a C function named "hello_world"
```

```
void ffihello_world(char *a, unsigned int alen,
                    char *b, unsigned int blen)
{
    printf("Hello World");
}
// The C function that we are calling
```

Pancake’s Memory

We can intialize stack allocated local variables using the following syntax:

```
var foo = 1;      // Initializing a variable "foo"
```

Pancake's Memory

We can also store and load bytes, or words, from the heap:

```
var heap_addr = @base;
// "@base" denotes the base of the heap
strb heap_addr, 1;
// Storing the literal "1" onto the heap at heap_addr
var foo = ldb heap_addr;
// Loading the value at heap_addr into foo
```

Pancake's Memory

We can also store and load bytes, or words, from the heap:

```
var heap_addr = @base;
// "@base" denotes the base of the heap
strb heap_addr, 1;
// Storing the literal "1" onto the heap at heap_addr
var foo = ldb heap_addr;
// Loading the value at heap_addr into foo
```

Pancake's Memory

We can also store and load bytes, or words, from the heap:

```
var heap_addr = @base;
// "@base" denotes the base of the heap
strb heap_addr, 1;
// Storing the literal "1" onto the heap at heap_addr
var foo = ldb heap_addr;
// Loading the value at heap_addr into foo
```

Pancake's Memory

We can also store and load bytes, or words, from the heap:

```
var heap_addr = @base;
// "@base" denotes the base of the heap
strb heap_addr, 1;
// Storing the literal "1" onto the heap at heap_addr
var foo = ldb heap_addr;
// Loading the value at heap_addr into foo
```

Example Pancake Code

```
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;
    }
}
```

Section 4

Case Study

So how have we used Pancake?

We have implemented the following Pancake components on the seL4 Device Driver Framework (sDDF):

- Serial Driver for the Freescale i.MX 8M Mini quad SoC.
- Serial Driver multiplexer.
- Ethernet Multiplexer for an Ethernet Driver written in C.

Related Posters

For more information on the sDDF please see:

”Secure, High-Performance I/O” by Lucy Parker

For more information on MicroKit please see:

”Verifying seL4 MicroKit” by Mathieu Paturel

How do we use Pancake?

basis_ffi.c

hello_world.pk

- Set up Pancake's memory regions.

How do we use Pancake?

basis_ffi.c

hello_world.pk

- Set up Pancake's memory regions.
- Initialise system.

How do we use Pancake?

basis_ffi.c

hello_world.pk

- Set up Pancake's memory regions.
- Initialise system.
- Jump into Pancake.

How do we use Pancake?

basis_ffi.c

hello_world.pk

- Set up Pancake's memory regions.
- Initialise system.
- Jump into Pancake.
- Handle FFI calls.

How do we use Pancake?

basis_ffi.c

- Set up Pancake's memory regions.
- Initialise system.
- Jump into Pancake.
- Handle FFI calls.

hello_world.pk

- This is our Pancake code.

How do we use Pancake?

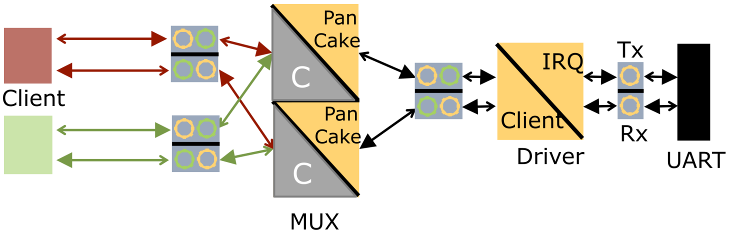
basis_ffi.c

- Set up Pancake's memory regions.
- Initialise system.
- Jump into Pancake.
- Handle FFI calls.

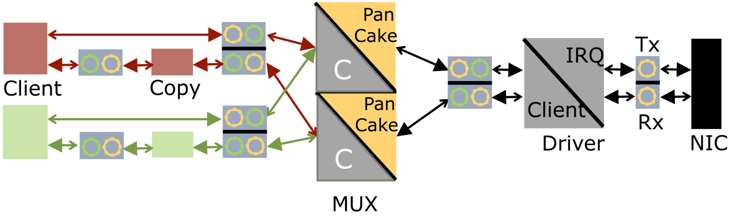
hello_world.pk

- This is our Pancake code.
- The Pancake compiler will output an assembly file.

Serial Driver and Multiplexer



Ethernet Multiplexer

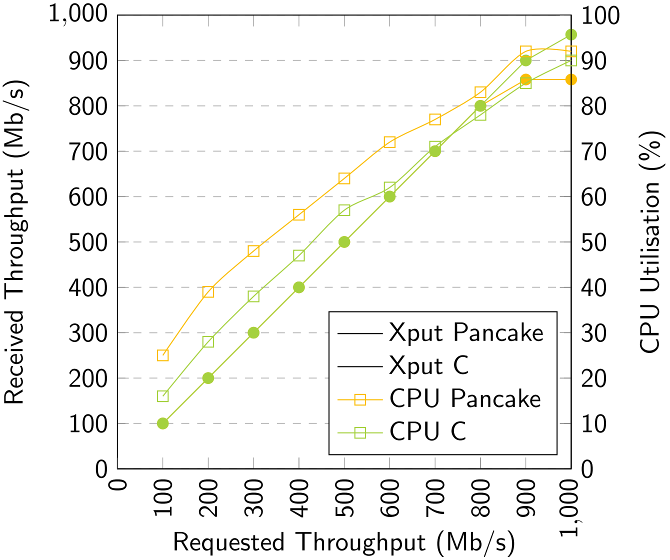


Issues we encountered

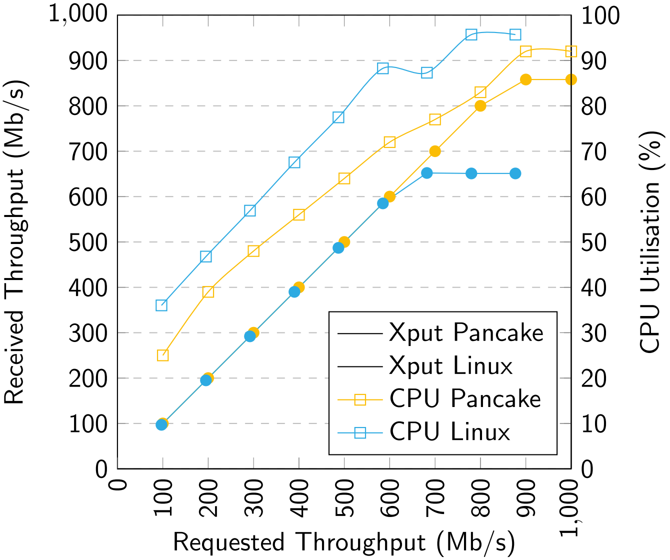
Due to Pancake currently being in the early stages of development, there were a few hurdles to overcome:

- Shared memory support.
- Memory management.
- Pancake entry points.
- Exiting Pancake.

Comparison against native C



Comparison against Linux



Section 5

Future Work

Future Work

Current Work:

- Shared Memory Semantics.
- Interaction Tree Semantics.
- Verification of Pancake progams.

Future Work:

- Decompilation into logic.

Section 6

Q&A