Running the Manual

An Approach to High-Assurance Microkernel Development

Philip Derrin

philip.derrin@nicta.com.au

Australian Government

Department of Communications, Information Technology and the Arts

Australian Research Council

NICTA Members

Department of State and Regional Development

NICTA Partners

RUNNING THE MANUAL

An Approach to High-Assurance Microkernel Development

Philip Derrin

philip.derrin@nicta.com.au

Or:

How to get Kernel Hackers to Work With Formal Methods

seL4 Project: Kevin Elphinstone, Philip Derrin, Dhammika Elkaduwe

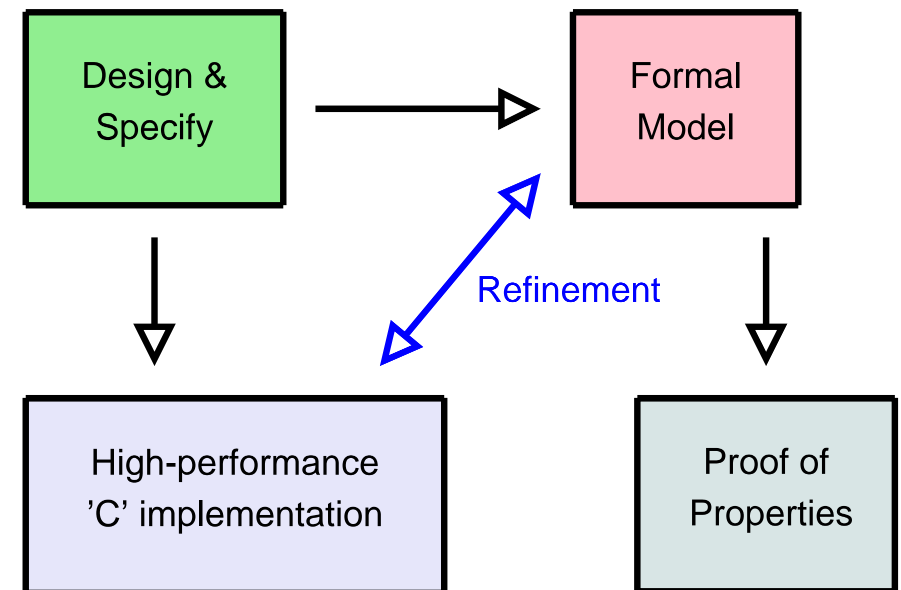
L4.verified Project: Gerwin Klein, David Cock, Thomas Sewell, Rafal Kolanski

NICTA Partners

NATIONAL ICT AUSTRALIA LIMITED

BACKGROUND

- seL4 Project
  - Develop a new microkernel API
  - Address the known deficiencies of L4
  - Provide a specification for formal verification
- L4.verified Project
  - Formally model the seL4 API
  - Prove properties of that API
- Goal
  - Develop a verified, high-performance microkernel implementation



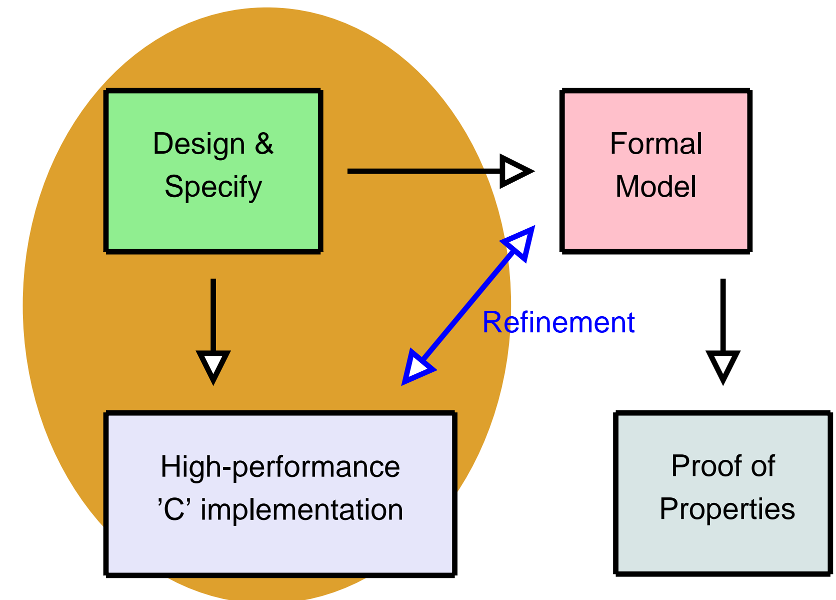
KERNEL DEVELOPMENT TECHNIQUES

Natural language specification:

- ➜ Easy to read and write
- ➜ Prone to ambiguity and incompleteness
- ➜ Fails to expose implementation issues

Low-level implementation (C, C++):

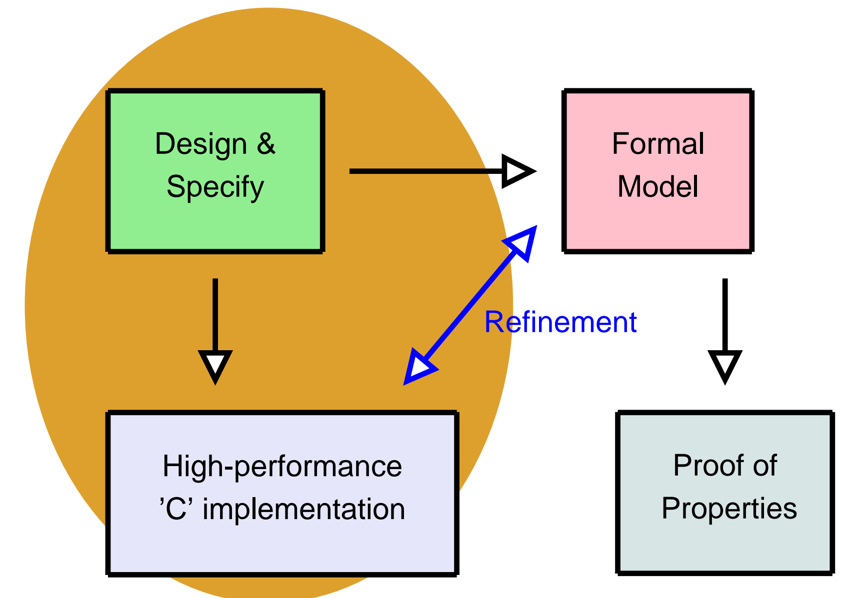
- ➜ Utility of the API can be evaluated
- ➜ Exposes implementation issues
- ➜ Performance is everything
  - ➜ Common code paths must be carefully hand-tuned
  - ➜ Significant penalty for poorly optimised code
- ➜ 5–10,000 lines of C and assembly code
- ➜ Tends to diverge from informal specifications



Kernel Development Techniques

Iterative prototype development:

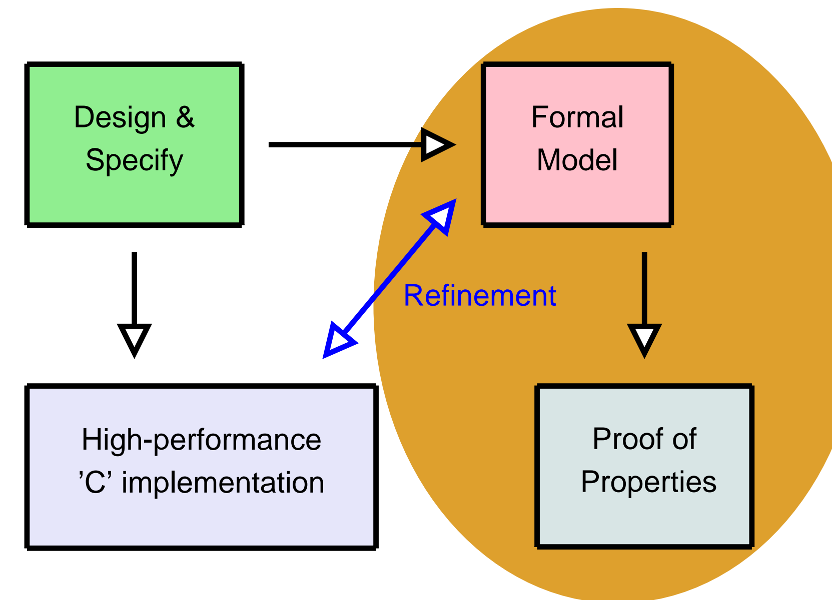
- ➜ Specify and implement in parallel
- ➜ Debugging prototypes is a significant fraction of development time
  - ➜ Bugs are likely
  - ➜ Debugging support is limited
- ➜ Difficult to keep the design and the prototype consistent



KERNEL FORMALISATION TECHNIQUES

Abstract Models:

- Unambiguous, but may still be incomplete
- Risks omitting details that are critical for kernels
  - Ease (or possibility) of implementation on bare metal
  - Issues that may impact performance
  - Utility of API for building systems



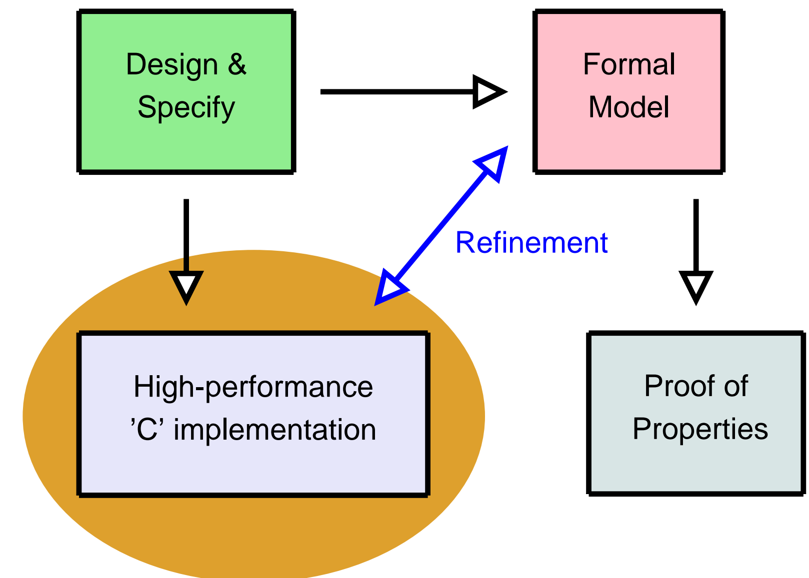
Verification of low-level code:

- Difficult to prove that a real implementation conforms to the formal model
- L4.verified pilot project demonstrated that this is difficult

KERNEL FORMALISATION TECHNIQUES

High-level implementation (Haskell, Standard ML):

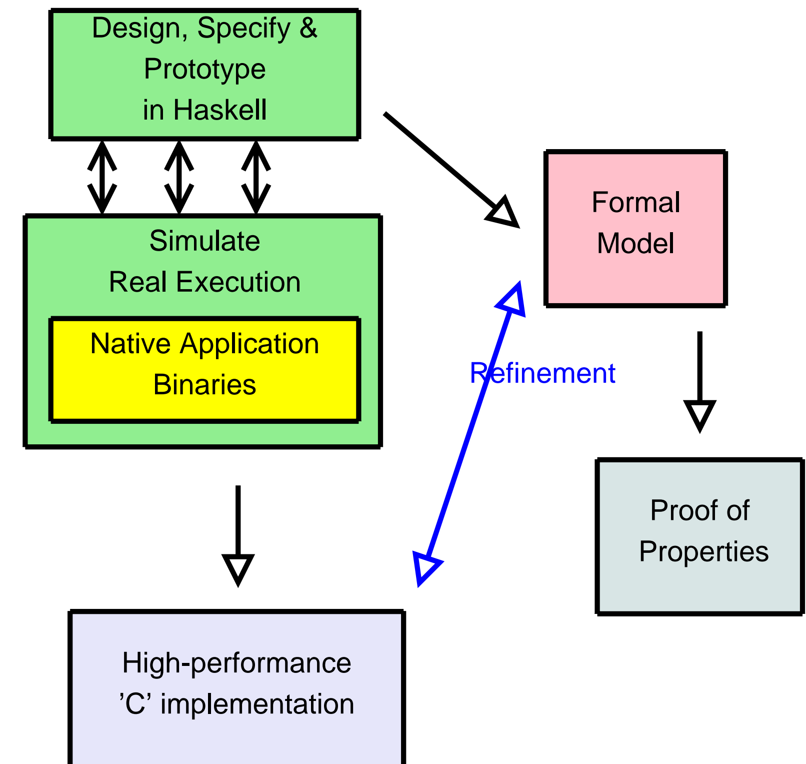
- ➜ Exposes performance and implementation issues
- ➜ Easier to reason about than C
- ➜ Language limitations and dependencies can cause problems
- ➜ May be difficult or impossible to attain high performance



Our Approach

Specify and Implement in Haskell:

- ➜ Executable prototype attached to a simulator
  - ➜ Runs user-level client code
  - ➜ Easy to evaluate utility
  - ➜ Exposes *most* implementation issues
- ➜ Implementation is deferred until the API is stable
- ➜ Semi-formal specification
  - ➜ High level language
  - ➜ Pure functional semantics
  - ➜ Easy to formalise
- ➜ Feedback between kernel developers and formal methods team



EXPECTATIONS

- Advantages:
  - Less low-level debugging work than a real implementation
  - Pure functional semantics ease formalisation
  - Exposes API usability issues via simulation
  - Permits rapid evaluation of design alternatives

NATIONAL ICT AUSTRALIA LIMITED

EXPECTATIONS

- Advantages:
  - Less low-level debugging work than a real implementation
  - Pure functional semantics ease formalisation
  - Exposes API usability issues via simulation
  - Permits rapid evaluation of design alternatives
- Limitations:
  - Relies on experienced kernel developers to foresee performance issues

KERNEL MODEL

Overview:

① Event processing

② Kernel state

③ Error handling

④ Simulation

EVENT PROCESSING

Events:

- ➜ The kernel we are modelling is event-driven
- ➜ Hardware notifies the kernel of events
- ➜ Kernel responds by manipulating kernel and hardware state

```
kernel ::  Event -> System -> System
```

EVENT PROCESSING

Events:

- The kernel we are modelling is event-driven
- Hardware notifies the kernel of events
- Kernel responds by manipulating kernel and hardware state

```
kernel ::  Event -> System -> System
```

- Event types:
  - System calls
  - Virtual memory access faults
  - Interrupts
  - Exceptions

EVENT PROCESSING

Kernel code is in a monad:

➜ Encapsulates the states of the kernel and hardware models

```
type Kernel = State System
kernel ::  Event -> Kernel ()
```

➜ Allows kernel code to appear procedural

➜ Gives us access to the Haskell monad template library

KERNEL STATE

Contents of the kernel state:

- ➜ Statically allocated global kernel data
  - ➜ Current thread
  - ➜ Scheduler queues
- ➜ Map of dynamically allocated kernel objects
  - ➜ Keys are physical addresses

KERNEL STATE

Dynamically allocated kernel objects:

- Several object types, depending on the kernel API:
  - Communication endpoints
  - Page tables
  - Capability table entries (CTEs)
  - Thread control blocks (TCBs)
  - Frames used to back virtual memory
- Functions of these objects are not relevant
- Properties of kernel object types
  - Physical size
  - Effects of system calls
  - Containers
    - These caused problems with the physical memory model

Physical Memory Model

Possible physical memory models:

① Untyped byte data

② Per-type structured storage

③ Unified structured storage

- ➜ Encapsulation
- ➜ Structure

PHYSICAL MEMORY MODEL

Array of bytes:

- ➜ Most realistic model
  - ➜ The kernel must eventually be refined to use this model
- ➜ Must convert between typed data and byte streams
- ➜ Forces data structures to be realistic
- ➜ Loses type information about stored objects
- ➜ Implementation in Haskell is painful

Physical Memory Model

Per-type Structured Storage:

- ➜ One array or map per kernel object type
- ➜ Easy to implement
- ➜ Need to demonstrate that we never use the same physical address for two different objects
  - ➜ Difficult to test using this model
  - ➜ Objects *do* overlap, when they are stored in containers (like CTEs in TCBs)

Physical Memory Model

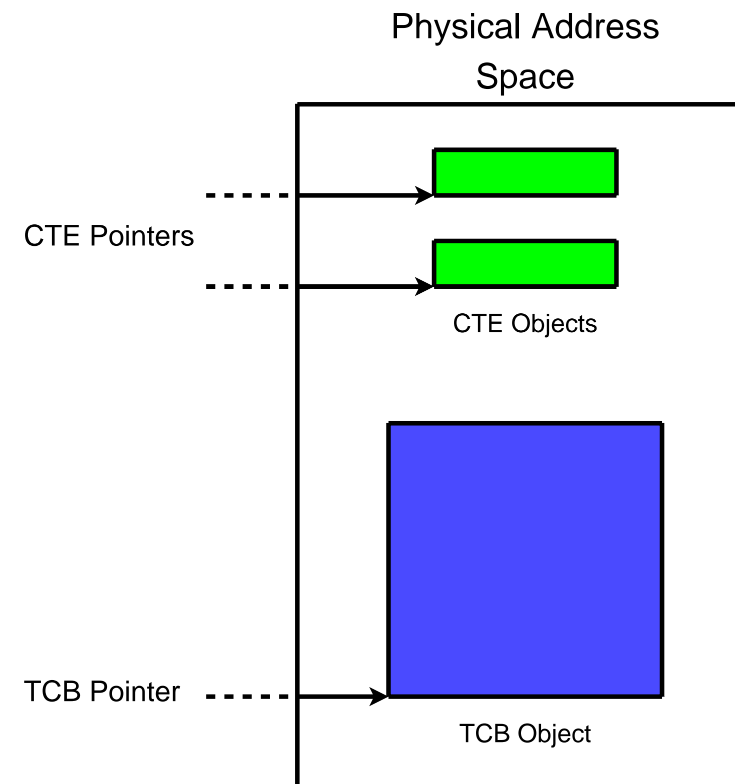
Unified Structured Storage:

- Objects must be encapsulated so they are all the same type
- Universal type
  - Caused difficulties when adding architecture-specific object types
- `Dynamic` containers
  - Easier to use, harder to formalise
  - Currently using this method, but could now switch back
- Need to choose an appropriate structure

Physical Memory Model

Map or Array:

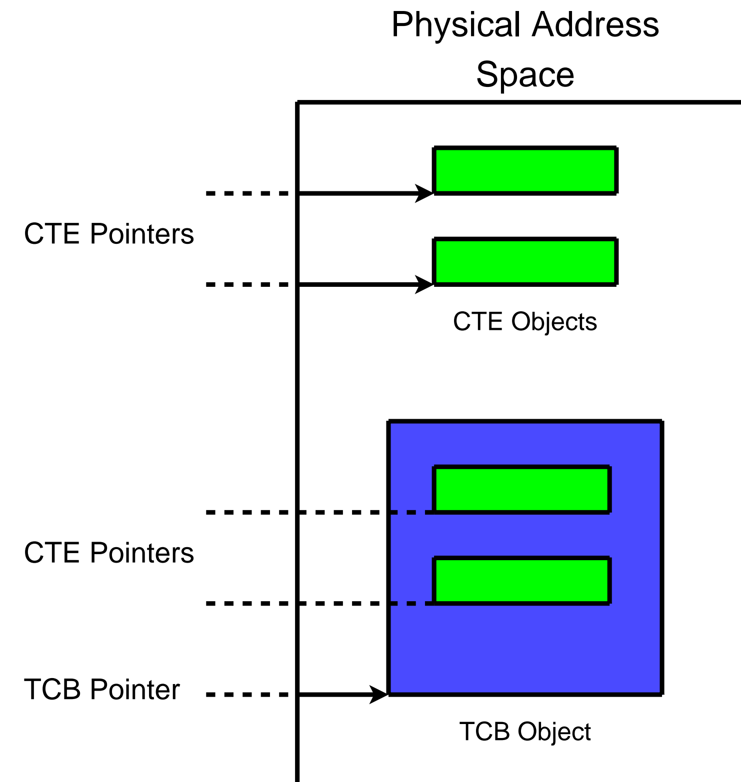
- Key or index is the physical base address
- One seL4 object type is sometimes contained in another
- Kernel accesses inner objects, unaware of the containers
- This doesn't work



Physical Memory Model

Map or Array:

- Key or index is the physical base address
- One seL4 object type is sometimes contained in another
- Kernel accesses inner objects, unaware of the containers
- This doesn't work

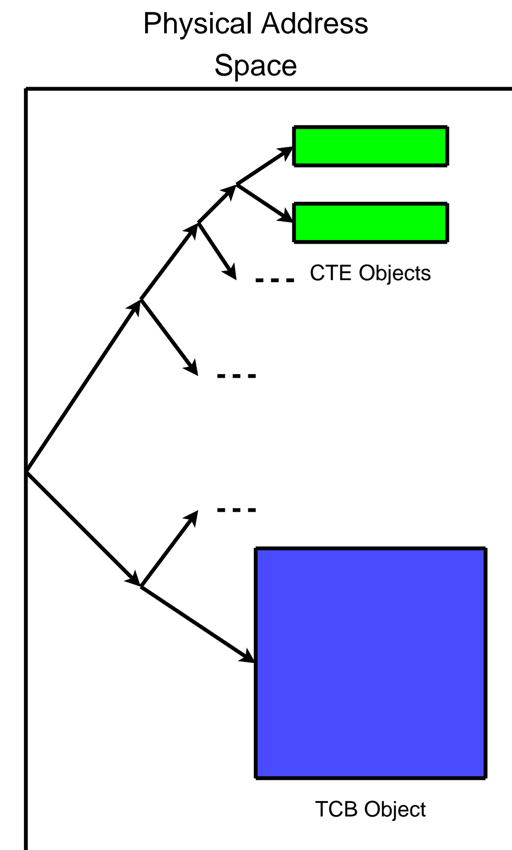


NATIONAL ICT AUSTRALIA LIMITED

PHYSICAL MEMORY MODEL

Binary tree indexed by physical address:

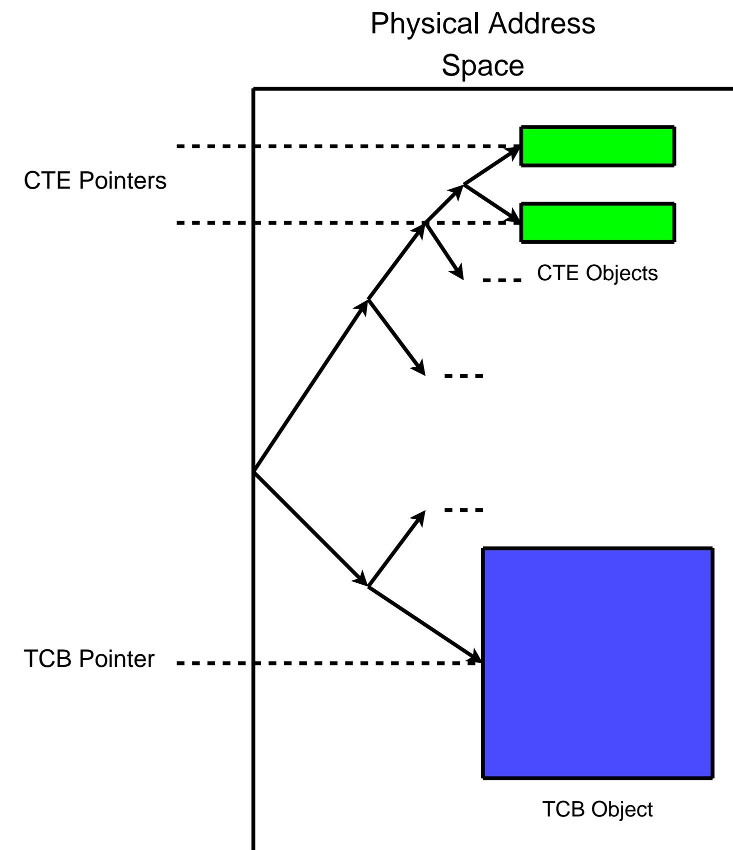
- ➜ Each address bit chooses a branch of the tree
- ➜ Number of bits used from the address depends on object size
- ➜ Remaining unused address bits are required to be 0



PHYSICAL MEMORY MODEL

Binary tree indexed by physical address:

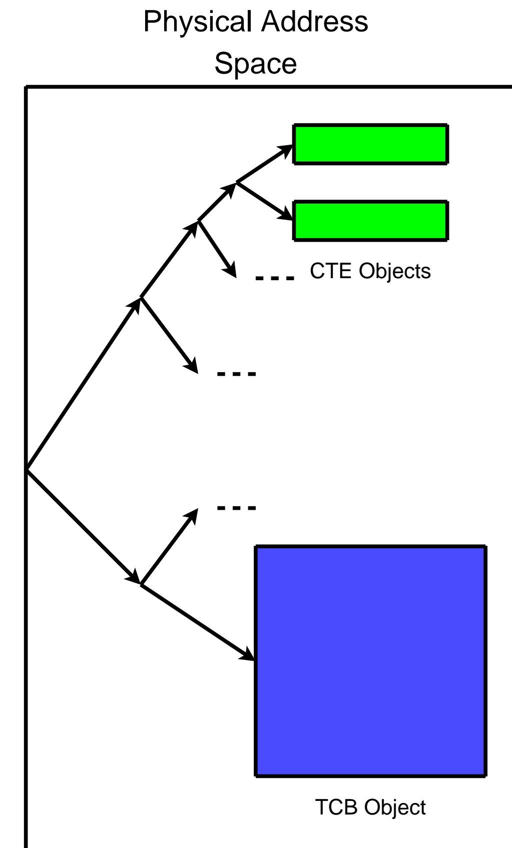
- ➜ Each address bit chooses a branch of the tree
- ➜ Number of bits used from the address depends on object size
- ➜ Remaining unused address bits are required to be 0



PHYSICAL MEMORY MODEL

Binary tree indexed by physical address:

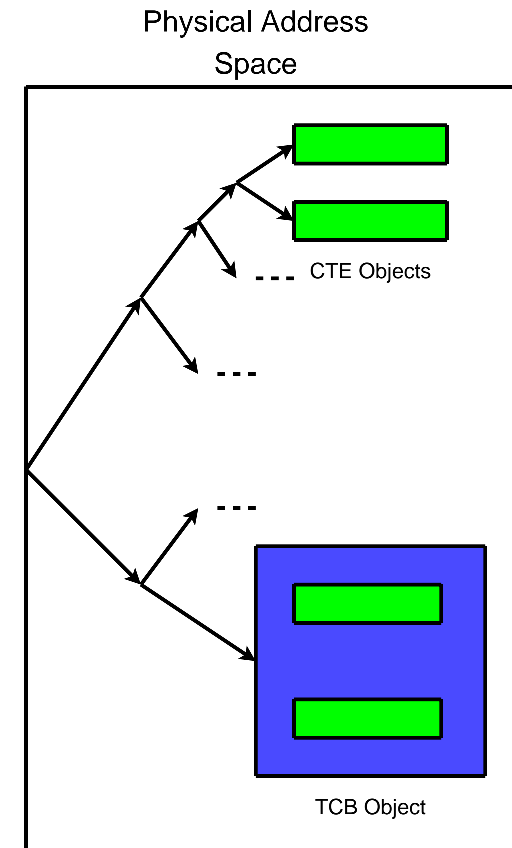
- ➜ Each address bit chooses a branch of the tree
- ➜ Number of bits used from the address depends on object size
- ➜ Remaining unused address bits are required to be 0
- ➜ Special case when a CTE is expected and a TCB is found instead



PHYSICAL MEMORY MODEL

Binary tree indexed by physical address:

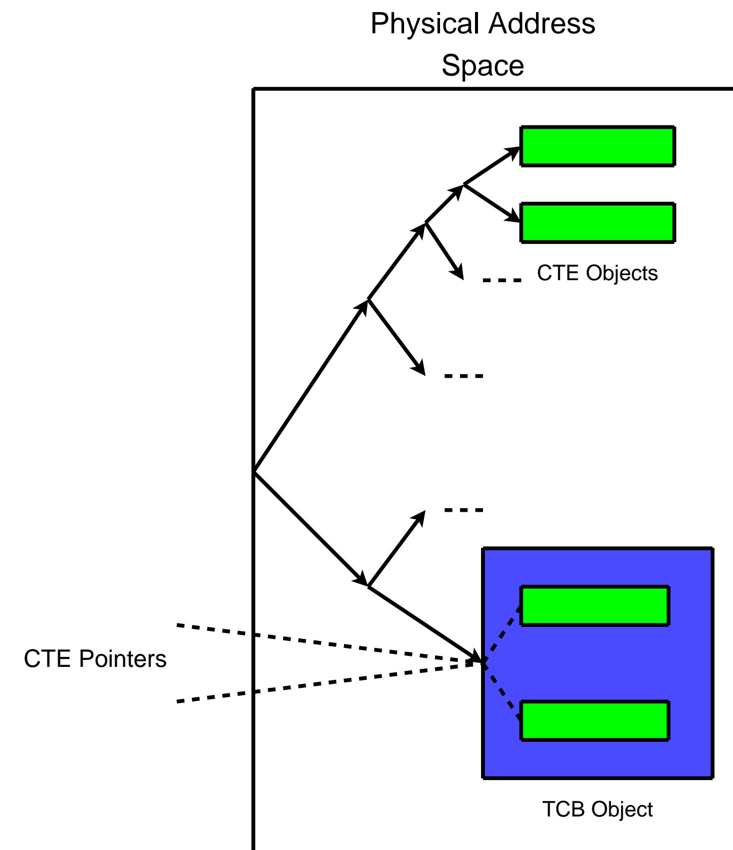
- Each address bit chooses a branch of the tree
- Number of bits used from the address depends on object size
- Remaining unused address bits are required to be 0
- Special case when a CTE is expected and a TCB is found instead



PHYSICAL MEMORY MODEL

Binary tree indexed by physical address:

- ➜ Each address bit chooses a branch of the tree
- ➜ Number of bits used from the address depends on object size
- ➜ Remaining unused address bits are required to be 0
- ➜ Special case when a CTE is expected and a TCB is found instead



Errors and Faults

Aborting an Operation:

- User level programs may cause error conditions:
  - Providing invalid system call arguments
  - Invoking kernel objects that don't exist
  - Asking the kernel to traverse an invalid user-managed data structure
- Kernel must abort the current operation and handle the error
- To support this, we use the ErrorT monad transformer

```
type KernelF e = ErrorT e Kernel
```

- Separates kernel code according to what sort of errors, if any, it can encounter

ERRORS AND FAULTS

Classes of Error:

- System call errors
  - Can be handled by the caller
  - Kernel replies to the caller with an error code
- Faults
  - Cannot be handled by the current thread
  - Kernel sends a notification to a nominated user-level handler
- Missing capabilities or invalid capability tables
  - Handling depends on the context
  - May cause system call errors or faults, or a silent failure

Errors and Faults

Classes of Error:

- ➜ System call errors
  - ➜ Can be handled by the caller
  - ➜ Kernel replies to the caller with an error code
- ➜ Faults
  - ➜ Cannot be handled by the current thread
  - ➜ Kernel sends a notification to a nominated user-level handler
- ➜ Missing capabilities or invalid capability tables
  - ➜ Handling depends on the context
  - ➜ May cause system call errors or faults, or a silent failure
- ➜ Fatal errors
  - ➜ Caused by kernel bugs or invalid states
  - ➜ Modelled by evaluating `undefined`
  - ➜ Verification must prove that these never occur

USER-LEVEL SIMULATION

Generating events:

① Hand-craft lists of events

- ➜ Difficult to construct realistic event sequences
- ➜ In a real system, events depend on kernel's actions

② Capture events from a running system

- ➜ Only makes sense if source and tested systems are similar
- ➜ Event sequence is independent of the kernel's actions

③ Generate events using a simulator

USER-LEVEL SIMULATION

- ➜ Simple assembly language in Haskell
- ➜ Similar to a RISC CPU
- ➜ Difficult to write non-trivial programs

```
pingThread :: UserText
pingThread = [
        Move AR0 R0,
        LoadImmediate 0 R1,

        Move R0 AR0,
        Move R1 AR1,
        DebugPrintf "Ping %" [R1],
        Syscall SysSendIPC,
        ArithmeticI R1 (+1) R1,
        Move R0 AR0,
        Syscall SysReceiveIPC,
        Branch (-7)
        ]
```

NATIONAL ICT AUSTRALIA LIMITED

USER-LEVEL SIMULATION

- Haskell FFI and existing external simulators
  - M5 Alpha simulator
  - Locally-developed ARMv6 simulator
- Allows execution of compiled user-level programs
- Easier to write or port complex test programs

```
void ping_thread(seL4_Capability ep)
{
  int i=1;
  seL4_Word sender;
  seL4_MessageInfo tag;
  seL4_Wait(ep, &sender, &tag);

  while(1) {
    printf("Ping %x <-> ",i);
    tag.length = 0;
    tag.label = i++;
    seL4_SendWait(ep, ep,
                  &sender, &tag);
  }

  return 0;
}
```

NATIONAL ICT AUSTRALIA LIMITED

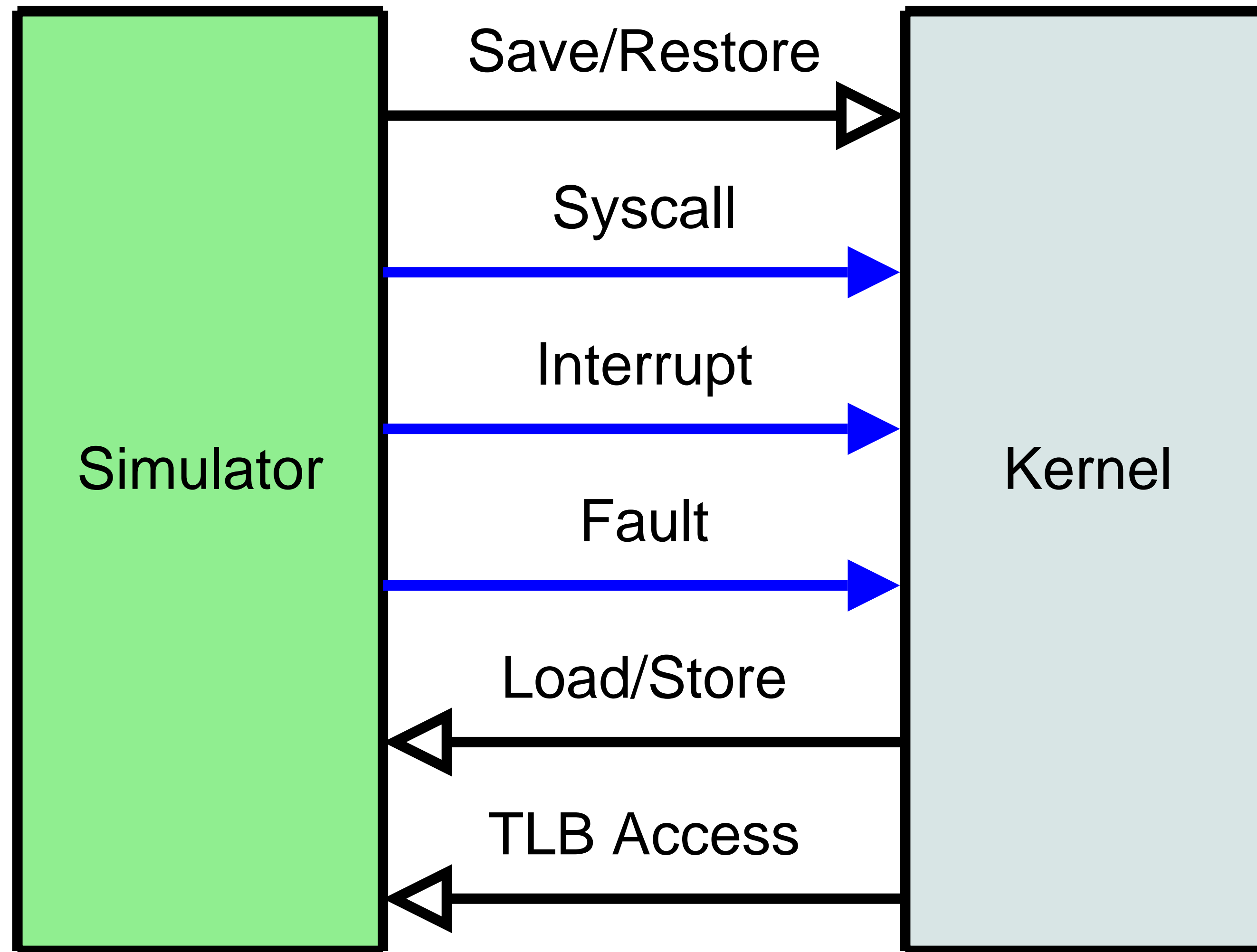
USER-LEVEL SIMULATION

Abstract hardware model:

- ➜ Interface is simplified and platform-independent:
  - ➜ Register set, including registers used for specific purposes
  - ➜ Hardware-defined data types
  - ➜ Virtual memory management
  - ➜ User level simulator
- ➜ Instances for specific hardware platforms

NATIONAL ICT AUSTRALIA LIMITED

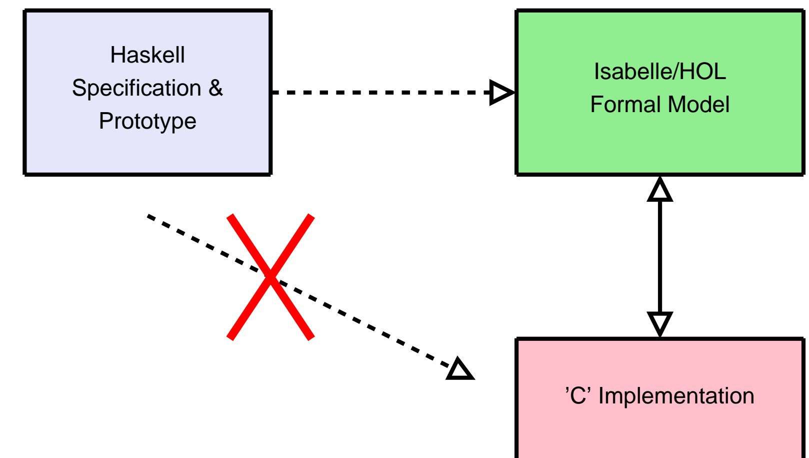
SIMULATOR INTERFACE



FORMALISATION

Translation to Isabelle/HOL:

- ➜ Translation was initially manual
- ➜ Now mostly automatic
- ➜ We take a pragmatic approach to translation
  - ➜ Simple python script
  - ➜ Entirely text processing; no real analysis of the Haskell
  - ➜ To simplify formalisation, we are limited to a sub-set of Haskell
- ➜ The translated Isabelle/HOL is the real formal specification, and will be refined into the final implementation



HASKELL VS. HOL

Termination:

- ➜ Haskell functions do not necessarily terminate
- ➜ HOL requires that we prove that all functions terminate
- ➜ We could use HOLCF to avoid this, but termination is a useful property

HASKELL VS. HOL

Termination:

- ➜ Haskell functions do not necessarily terminate
- ➜ HOL requires that we prove that all functions terminate
- ➜ We could use HOLCF to avoid this, but termination is a useful property
- ➜ We have proved termination for every kernel function
  - ➜ Only had to cheat once

HASKELL VS. HOL

Termination:

- ➜ Haskell functions do not necessarily terminate
- ➜ HOL requires that we prove that all functions terminate
- ➜ We could use HOLCF to avoid this, but termination is a useful property
- ➜ We have proved termination for every kernel function
  - ➜ Only had to cheat once

Monads:

- ➜ State and ErrorT have multiple type parameters
- ➜ It's inconvenient to formalise them completely in HOL
- ➜ We only formalise the instances that we need

EXPERIENCE

Evolution of the API design:

- ➜ Explored three major changes relative to L4:
  - ➜ Access control mechanisms for IPC
  - ➜ Kernel resource management
  - ➜ Kernel invocation mechanisms

EXPERIENCE

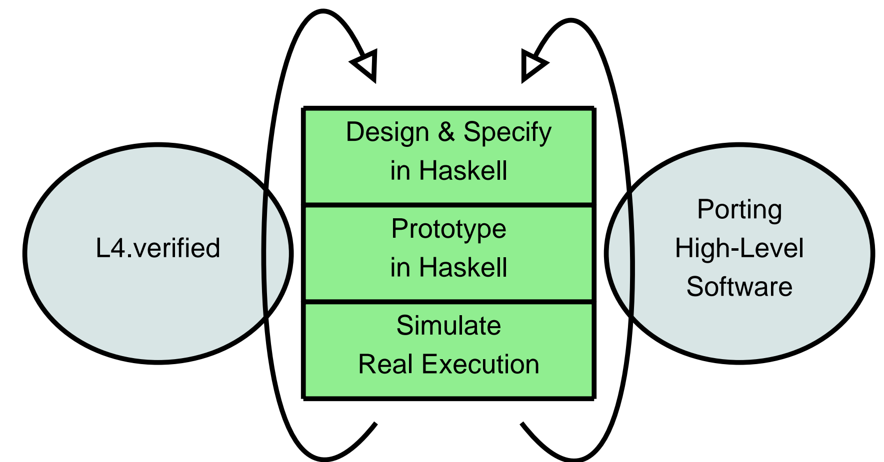
Evolution of the API design:

- ➜ Explored three major changes relative to L4:
  - ➜ Access control mechanisms for IPC
  - ➜ Kernel resource management
  - ➜ Kernel invocation mechanisms
- ➜ No need to implement a complete kernel before experimenting
  - ➜ First IPC test ran without a virtual memory model
  - ➜ First kernel resource management tests used single-level tables
  - ➜ Still working on ARM hardware-defined page tables

EXPERIENCE

Parallel Development:

➜ Translation to Isabelle/HOL started relatively early



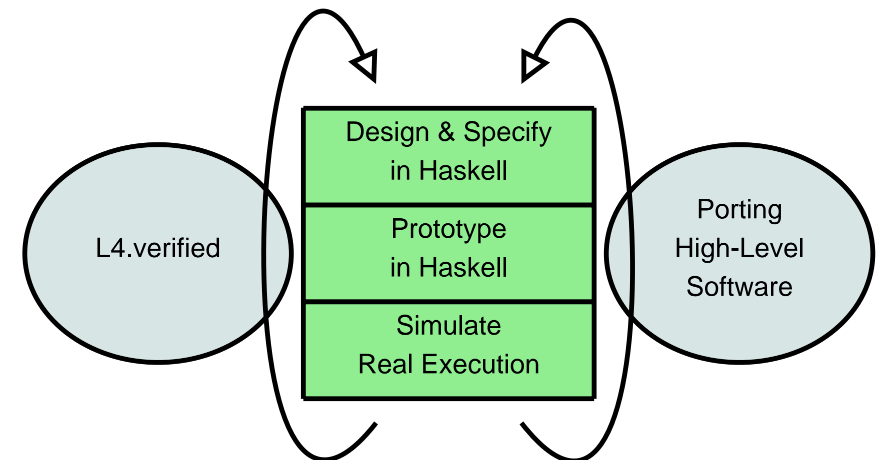
Verification team identifying conceptual and verification issues.

Kernel team identifying conceptual and potential performance issues.

EXPERIENCE

Parallel Development:

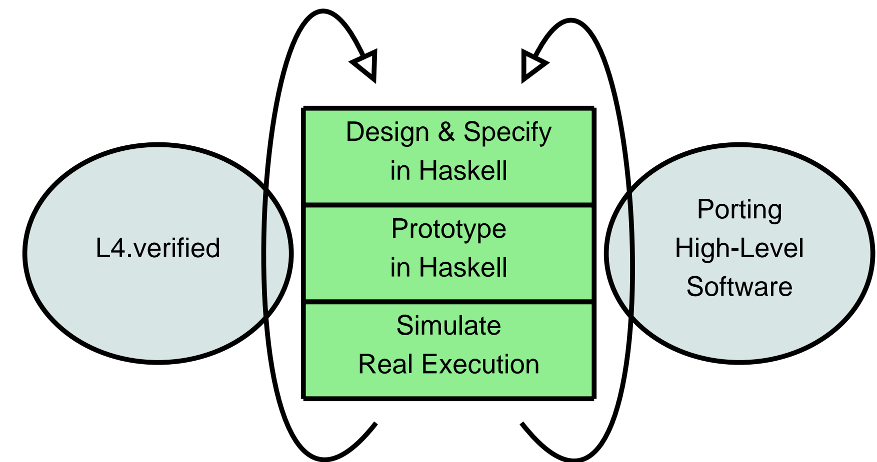
- ➜ Translation to Isabelle/HOL started relatively early
- ➜ Formalisation process has detected API problems
  - ➜ Unbounded IPC send operation



EXPERIENCE

Parallel Development:

- ➜ Translation to Isabelle/HOL started relatively early
- ➜ Formalisation process has detected API problems
  - ➜ Unbounded IPC send operation
- ➜ Simulation has also revealed API problems
  - ➜ Missing operation: Identify
  - ➜ Implemented, tested, formalised within a few hours



DEMO

QUESTIONS

Questions?