The Clustered Multikernel:
An Approach to Formal Verification of Multiprocessor OS Kernels

Michael von Tessin

PhD Student
NICTA and University of New South Wales
Sydney, Australia

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

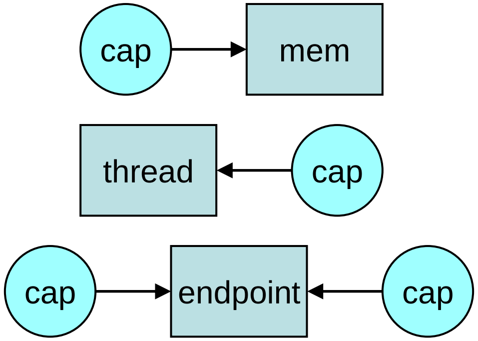
NICTA Funding and Supporting Members and Partners

Introduction

- OS kernel is a critical software component in computer systems
- building secure, safe and reliable computer systems is facilitated by having strong kernel correctness guarantees

→ **formal verification** down to implementation level

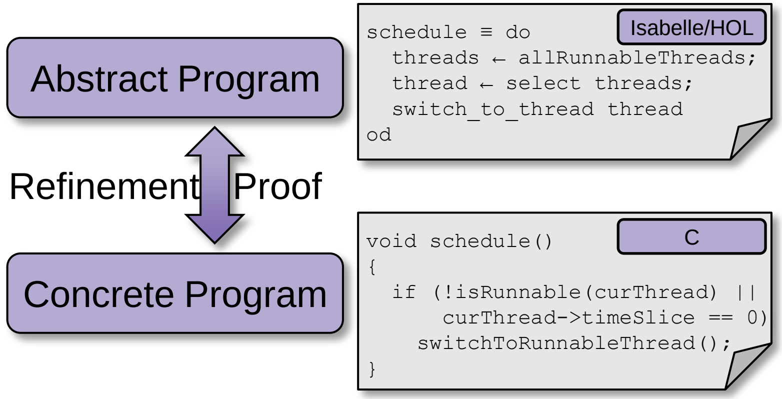
- **seL4** (secure embedded L4) **microkernel**:
  - provides strong isolation between components
  - allows fine-grained controlled communication and resource management via capabilities
  - C implementation is formally verified
  - property: **functional correctness**



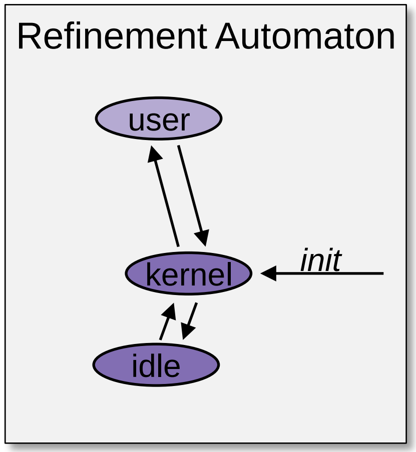
Refinement Background

Property: **functional correctness**

- proved by **refinement**
- **abstract**: specification
  - abstract program modifies abstract state
- **concrete**: implementation
  - concrete program modifies concrete state



- refinement automaton:
  - non-deterministic finite state automaton
  - **initialisation function** sets up initial state (corresponds to **bootstrapping phase** of kernel)
  - **events** trigger **transitions** between **states** (models the **runtime phase** of kernel)



- refinement proof:
  - consists of:
    1. **correspondence** proofs, which sometimes require
    2. **invariant** proofs
  - **transfers** theorems proved on the abstract level down to the concrete level → sufficient to prove theorems on abstract level

Motivation

- L4.verified approach has no concurrency in the model:
    1. able to avoid preemption-induced concurrency:
        - no preemption in kernel
        - except from two well-defined preemption points
        - instead of doing a stack switch, kernel saves state as continuation and exits
    2. able to avoid hardware concurrency:
        - device drivers outside the kernel (standard microkernel approach)
        - **only support uniprocessor systems**
- whole world is going **multicore** (even in embedded systems)

→ need for **verified multiprocessor kernels** arises

- aim: want to have a multiprocessor version of seL4 with the same **functional correctness** guarantees
- want to leverage as much of the uniprocessor proof as possible
    - L4.verified total effort: ~25py / 200 kLOC of proof

Challenges?

Verification Complexity

- we are hit with full concurrency of multiple CPUs
- proof needs to cover all possible "conceptual scenarios" which can arise from concurrent execution
- verification complexity depends on:
  - program complexity
  - **property** we want to prove
  - **state** we share (if concurrency is involved)

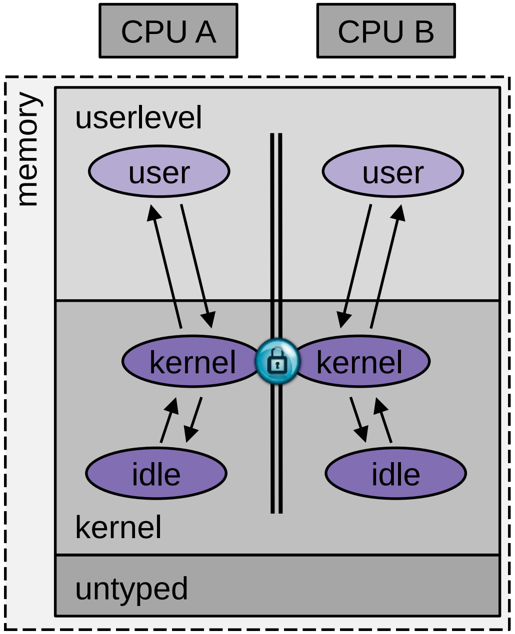
- mitigation techniques:
  - make proofs modular (e.g. rely-guarantee, ownership principle)
    - only works if modeled system can somehow be viewed in a modular way
  - make the system modular → componentise it
  - microkernels cannot be componentised

→ **approach**:
  - reduce the number of "conceptual scenarios" to a minimum
  - by avoiding *complexity potentially introduced by parallelism*

Multiprocessor Kernel Designs

There are two fundamental ways to avoid complexity potentially introduced by parallelism:

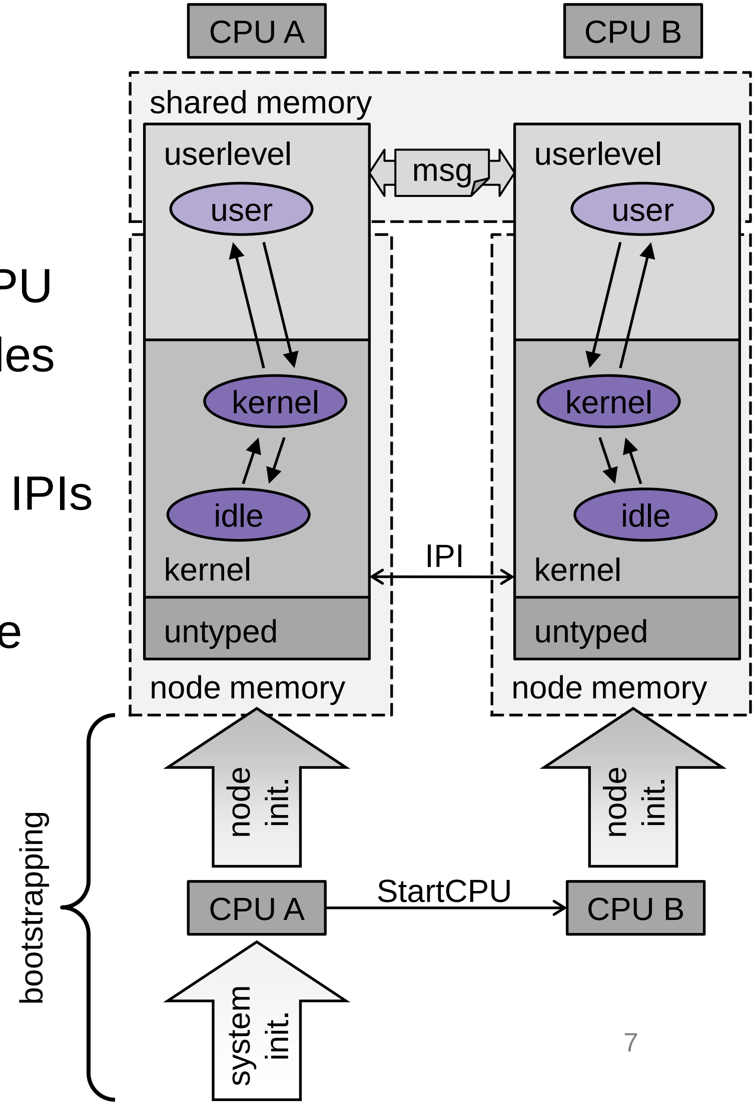
1. avoid parallelism itself (run things sequentially):
   - solution: ***big lock*** around the whole kernel
   - 👍 existing uniprocessor userlevel applications can be run unmodified and automatically benefit from the power of multiple CPUs
   - 👎 low scalability



Multiprocessor Kernel Designs

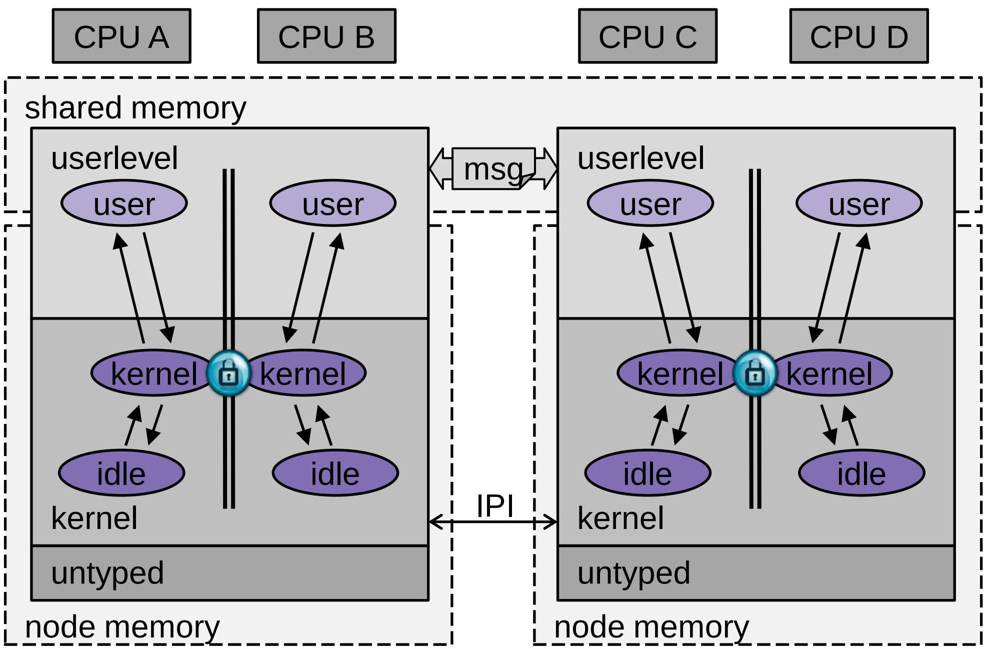
There are two fundamental ways to avoid complexity potentially introduced by parallelism:

2. establish independence (avoid sharing):
   - solution: restricted ***multikernel*** design
   - run one *node* of uniprocessor seL4 per CPU
   - kernel memory is partitioned between nodes
   - static region of shared userlevel memory
   - communication via userlevel memory and IPIs
   - high scalability
   - userlevel applications must be node-aware
   - no flexible kernel-memory usage between nodes



The Clustered Multikernel

- now we have two designs:
  - **big-lock kernel**: high flexibility, low scalability
  - **multikernel**: low flexibility, high scalability
- combine them: ***clustered multikernel***
  - like **multikernel**, but a node can span more than one CPU
  - within a node, kernel data is protected by a **big lock**
  - CPUs can be freely assigned to nodes



performance-optimisation opportunities:
- cluster of cores within a CPU
- NUMA-aligned clusters
- clustering for systems with “islands of cache coherence”
- clustering along performance-isolation boundaries

implementation: **seL4::CMK**

Lifting seL4's Refinement Proofs

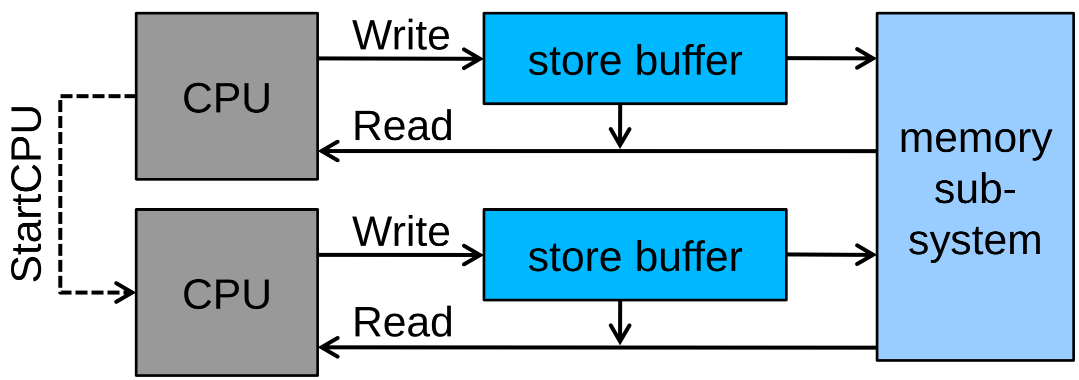
- lifting proofs = reusing proved theorems in a more generic context
  - e.g., a proved hoare triple over a kernel-internal function can be directly reused in the clustered-multikernel proof if we prove that this particular function is not exposed to concurrency

- refinement lifting proof consists of:
  - abstract specification of seL4::CMK's code
  - model of a total-store-order (TSO) multiprocessor architecture
    - deal with weak memory ordering, memory fences
    - needed for inherently concurrent **bootstrapping phase** of the kernel
  - node-isolation proof
    - want to be able to reason about each node in isolation
    - show: for seL4::CMK, refinement holds for each node in isolation
  - within each node:
    - refinement automaton represents **runtime phase** of the kernel
    - lifting of the refinement automaton into a **parallel composition** of itself

- specifications and proofs are machine-checked in Isabelle/HOL

TSO Multiprocessor Model

- challenges:
  - weak memory ordering and fences
  - in presence of CPUs starting up other CPUs (also nested)
  - integratable into L4.verified verification framework
- model:
  - operational model, inspired by the Cambridge x86-TSO model
  - 4 *high-level instructions*: Read, Write, MFENCE, StartCPU
- proof:
  - generic *sequential-semantics theorem* (MFENCEs, starting seq.)

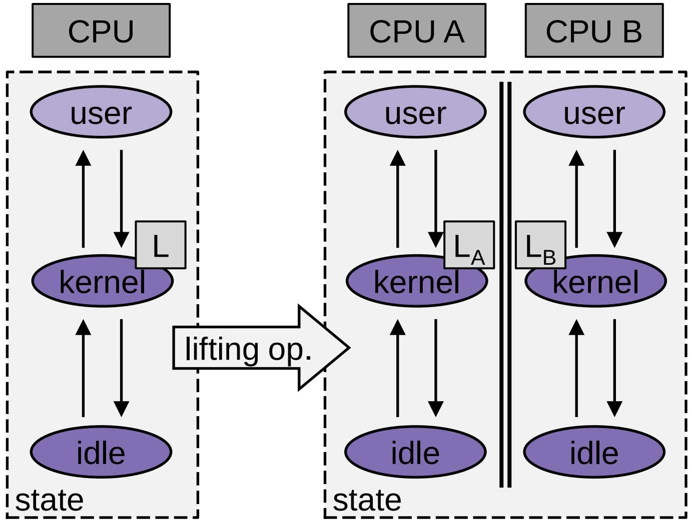


program order preserved	
R,R	✓
W,W	✓
R,W	✓
W,R	✗

Lifting into Parallel Composition

- lifting operation:
  – converts an arbitrary refinement automaton into parallel composition

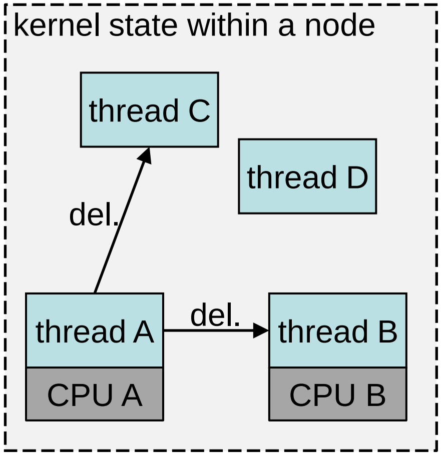
- lifting theorem:
  – “When applying the lifting operation to the refinement automata of both abstract and concrete levels of an arbitrary refinement proof:
    the **concrete parallel** refinement automaton
    refines the **abstract parallel** refinement automaton
    if the **original concrete** refinement automaton
    refines the **original abstract** refinement automaton.”



- transitions are interleaved non-det.
- original state is **shared** between CPUs
  – except for small subset which is **local**, i.e. duplicated for each CPU (L_A, L_B)
  – each CPU can modify the shared state and its own local state
- precondition:
  – original invariants need to be **splittable**
  – **unsplittable** ones proved manually
    - in seL4: valid pointer to currently running thread

Thread-Deletion Problem

- could not prove seL4's **unsplittable** invariants (did not hold)
- thread-deletion problem:
  - formally: CPU B's pointer to the currently running thread is not valid anymore if thread B is deleted by thread A running on CPU A
  - could not happen in uniprocessor case: only one thread currently running



- fix (add necessary coordination):
  - 2 new thread states ("current", "current req. inactive")
  - 8 new preemption points (before del./mod. of threads)
  - reason: event-based structure of seL4
    - specifically: no kernel-thread blocking allowed
  - changes small but invasive

  → increased proof complexity considerably

summary:

- specific to seL4, but likely to occur in other kernels as well
- a good example in showing the bug-finding abilities of theorem proving in general, and the lifting theorem in particular

Related Work

- Barrelfish:
  - **multikernel** OS designed for heterogeneous multiprocessing
  - follows a distributed-system approach by keeping kernel data structures local to a CPU or replicated on other CPUs
  - communication between nodes message-based, on userlevel

clustered kernels in the early 90s:

- Hurricane:
  - used clustering to improve data locality on large-scale NUMA machines
- Hive:
  - aimed at fault isolation between clusters

→ performed well for certain kinds of applications, but suffered from high complexity and unpredictable performance in general

- probably because they tried to hide clustering from userlevel and provide a single-system image

Conclusion and Future Work

Conclusion:

- implementation effort for seL4::CMK (diff. to seL4): ~**0.5 kLOC**
- the proof effort was ~**9 kLOC** (*conditions apply)
- not aware of a successful refinement proof of a multiprocessor kernel
- given a verified uniprocessor kernel, the clustered multikernel offers a way to achieve this with *relatively* low effort
  - compare **~0.5 kLOC** to seL4's code size of **~8.7 kLOC**
  - compare **~9 kLOC** to L4.verified's overall proof size of ~**200 kLOC**

Future Work:

- performance/scalability evaluation showing that the clustered multikernel is a "viable alternative" to a classical MP kernel:
  - a classical MP kernel (fine-grained locks/lock-free) would give us:
    1. good scalability, **and at the same time**
    2. flexible kernel-memory usage across CPUs
  - but for verification reasons, we restrict ourselves to a clustered multikernel where we only get a static tradeoff between (1) and (2)

  → want to show (benchmarks) that this is NOT a serious restriction

Questions?