The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels

Michael von Tessin

Ph.D.

March 2013

School of Computer Science and Engineering
Faculty of Engineering

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **von Tessin**

First name: **Michael** Other name/s:

Abbreviation for degree as given in the University calendar: **PhD**

School: **Computer Science and Engineering** Faculty: **Engineering**

Title:
The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels

Abstract 350 words maximum: (PLEASE TYPE)

The key software component of a computer system is the operating-system kernel. It always needs to be trusted because it runs in the CPU's privileged mode and therefore has access to all system components. Consequently, kernel correctness is crucial for secure, safe and reliable computer systems. Correctness can be improved by careful design, development and testing. However, this is not enough for kernels of high-assurance computer systems used in defence, aviation and the like.

Much stronger correctness guarantees can be obtained by formal verification of a kernel's implementation. In order to keep verification complexity at a manageable level, prior kernel verification research only targeted uniprocessor kernels. In other words, the current state of research restricts computer systems that require a verified kernel to running on one CPU/core. This is a problem because manufacturers are increasing computing power of their systems by adding more CPUs and cores.

In this thesis, we demonstrate that it is possible to extend a verified uniprocessor kernel to utilise multiple CPUs/cores and leverage the existing proofs to obtain a verified multiprocessor version of that kernel (under certain assumptions).

To this end, we introduce the *clustered multikernel*, a point in the design space of multiprocessor kernels. The main feature of this design is that it reduces concurrent data access to a minimum while offering a configurable trade-off between scalability and flexibility. Furthermore, we present a *conversion scheme* to convert a uniprocessor kernel into a clustered multikernel.

Based on this design, we contribute a *refinement lifting framework*, which lifts the converted kernel's functional-correctness proof such that it applies to the clustered-multikernel version. The support for handling the introduced concurrency is added to the existing verification framework in a non-intrusive way and accounts for TSO weak memory ordering.

We demonstrate the practicability of our approach by successfully applying it to seL4, a formally verified general-purpose microkernel. We show that this requires relatively low effort, compared to the kernel's initial verification.

All formal specifications and proofs are machine-checked in the theorem prover Isabelle/HOL.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signature: .....M. van Teeseling.....

Date: .....28/03/2013.....

Copyright Statement

Authenticity Statement

Abstract

The key software component of a computer system is the operating-system kernel. It always needs to be trusted because it runs in the CPU's privileged mode and therefore has access to all system components. Consequently, kernel correctness is crucial for secure, safe and reliable computer systems. Correctness can be improved by careful design, development and testing. However, this is not enough for kernels of high-assurance computer systems used in defence, aviation and the like.

Much stronger correctness guarantees can be obtained by formal verification of a kernel's implementation. In order to keep verification complexity at a manageable level, prior kernel verification research only targeted uniprocessor kernels. In other words, the current state of research restricts computer systems that require a verified kernel to running on one CPU/core. This is a problem because manufacturers are increasing computing power of their systems by adding more CPUs and cores.

In this thesis, we demonstrate that it is possible to extend a verified uniprocessor kernel to utilise multiple CPUs/cores and leverage the existing proofs to obtain a verified multiprocessor version of that kernel (under certain assumptions).

To this end, we introduce the *clustered multikernel*, a point in the design space of multiprocessor kernels. The main feature of this design is that it reduces concurrent data access to a minimum while offering a configurable trade-off between scalability and flexibility. Furthermore, we present a *conversion scheme* to convert a uniprocessor kernel into a clustered multikernel.

Based on this design, we contribute a *refinement lifting framework*, which lifts the converted kernel's functional-correctness proof such that it applies to the clustered-multikernel version. The support for handling the introduced concurrency is added to the existing verification framework in a non-intrusive way and accounts for TSO weak memory ordering.

We demonstrate the practicability of our approach by successfully applying it to seL4, a formally verified general-purpose microkernel. We show that this requires relatively low effort, compared to the kernel's initial verification.

All formal specifications and proofs are machine-checked in the theorem prover Isabelle/HOL.

Acknowledgements

I first want to thank my supervisor, Kevin Elphinstone, for giving me the opportunity to work on this PhD project and introducing me to the world of research. He taught me how to think scientifically and how to break down, analyse and present my work. His thorough feedback on my drafts helped me improve my writing greatly. He gave me a lot of freedom with regards to the focus and planning of my PhD and was always there when I asked him for feedback, advice and support.

I would also like to thank my professor, Gernot Heiser, for accepting me as a PhD student in such a prestigious research group. I enjoyed a highly inspiring and motivating yet relaxed work atmosphere. However, I would never have known the existence of this research group without Felix Rauch, who encouraged me to apply and helped me settle once I was there, and whom I want to thank for that.

Numerous people in our group have helped me over the past years and I want to take this opportunity to thank all of them. In particular, I am grateful to Gerwin Klein whom I could always ask for advice and feedback on the formal aspects of my work. I would also like to thank Thomas Sewell for helping me grapple with Isabelle/HOL proofs, and Rafal Kolanski for showing me all the necessary tricks to get my thesis typeset the way I wanted. Furthermore, I want to thank Leonid Ryzhyk and Nicholas FitzRoy-Dale for their valuable feedback on my thesis draft.

Finally, and most importantly, I want to thank my family. My utmost gratitude goes to my wife Gabi for her love and constant support throughout these years. Without it, and without her agreeing to move to Sydney, this thesis would not have been possible. I am also grateful to my daughters, Lena and Neva, whose joyful and lively nature always helped me switch off at the end of a working day.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The way we live today increasingly depends on computer systems. We routinely employ desktop computers to perform day-to-day tasks, we depend on embedded systems when driving a car or flying in an aircraft, and we indirectly use the servers of companies and governments that we interact with. We rely on these systems to be secure, safe and reliable.

Computer systems are built of hardware and software components. Security, safety and reliability of computer systems depend on the correctness of multiple components and on how these components interact with each other. For a desired security property, the subset of components that have to be trusted for the property to hold is called the *trusted computing base (TCB)*. Components outside the TCB are not critical and the security property still holds even if those components are faulty.

The key software component of a computer system is the operating-system (OS) kernel. It is always in the TCB because it runs in the CPU's privileged mode and therefore has access to all hardware components, which it multiplexes between software components. Therefore, OS-kernel correctness is crucial for secure, safe and reliable computer systems.

According to Hatton [Hat97], "good" software contains 6 bugs[1] per thousand lines of code (kLOC) on average and with "our best techniques" (such as careful design, development and testing) we can achieve 0.5–1 bugs per kLOC. However, this is not enough for kernels of high-assurance computer systems used—for example—in defence, aviation and the like.

Much stronger correctness guarantees can be obtained by formally verifying the correctness of a kernel's implementation. The history of kernel verification starts in the 70s and 80s [NBF+80, WKP80, Bev89], but none of these early attempts "produced a realistic kernel with full implementation proofs" [Kle09]. In the years after 2000, the topic attracted new interest: Verifying a kernel was part of several verification projects [HT05, TWV08, SDN+04, DDW09, DSS09, KEH+09, YH10]. The largest verified kernel is seL4 [KEH+09], an 8700-LOC general-purpose microkernel, which was verified down to the implementation level. "Verified" means there is a *functional-correctness* proof saying that the implementation adheres to a formal specification of the desired functionality.

In order to bring verification complexity down to a manageable level, the kernels mentioned above have two things in common: First, they are relatively small, in the order of a few kLOC. Second, their designs avoid concurrency within the kernel because concurrent software is very hard to reason about, as we will see in Section 1.2.

In-kernel concurrency is typically introduced by (1) switching between multiple threads of execution (kernel threads, interrupt handlers) or (2) running code in parallel

[1] A *bug* is a fault in the software as a result of a programming error. A bug can make the software behave in ways that were not intended (e.g. specified).

on multiple CPUs. The former can be overcome by making a kernel non-preemptible or event-based with well-defined preemption points. In order to deal with the latter, the kernels mentioned above are restricted to only support a single CPU (or core[2]).

In summary, the current state of research restricts high-assurance computer systems that require a verified kernel to running on one CPU. This is a problem because manufacturers are increasing the computing power of their systems by adding more CPUs and cores. The problem has existed in the server and desktop space for years and finds its way into the embedded world now.

Consequently, this thesis aims at developing an approach to formal verification of multiprocessor kernels.

1.1 Functional Correctness

There is a plethora of verification properties that can be proved about programs. There are lower-level properties such as termination, correctness of assertions and absence of null-pointer dereferences, overflows or exceptions. On the other side of the spectrum, we have higher-level properties such as isolation, integrity and *functional correctness*.

The latter is a very powerful verification property. It is proved by *refinement*, in which the implementation of a program refines an abstract specification of the desired functionality. In other words, it is proved that the program's behaviour conforms to what is specified.[3]

Abstract specifications are much smaller and easier to understand then actual program code, which reduces the probability of bugs. Moreover, specification bugs can be eliminated by additional proofs on the abstract level. A property proved on the abstract level is transferred by refinement to the implementation level, i.e. we know that it also holds in the implementation. This applies to so called *safety hyperproperties* [CS08], which include higher-level properties such as isolation and integrity. Proving such properties on the abstract level requires an order of magnitude less effort than proving them directly on the implementation level [KMG+11].

1.2 Verification Complexity

Formal verification frameworks are usually either based on a *model-checking* approach or a *theorem-proving* approach.

In model checking, the system is captured in a mathematical model with enumerable states and transitions. For a desired verification property, the framework then automatically explores all possible states (via transitions) in order to verify whether the property is satisfied in all of them. As such, the verification process is fully automatic once the model has been created. There are a variety of abstraction techniques in order to reduce the number of states that have to be checked. However, model checking still becomes intractable quickly for large state spaces. This is especially true if concurrency is involved because the number of states grows exponentially with the degree of parallelism. Model checking is often used for verifying correctness of protocols, e.g. cache coherence protocols implemented in hardware [PD97] or synchronisation protocols in software. As

[2]Throughout this thesis, we use the term *CPU* to refer to the logical unit of execution, e.g. a unicore processor or a core within a multicore processor. As such, we also use the term *multiprocessor* instead of *multicore*, unless we explicitly need to distinguish between them.

[3]Usually there are assumptions such as a correct compiler, correct inline-assembly code, correct verification framework etc. Sometimes, there are also additional intermediate specification levels.

these protocols have a limited number of states, model checking is tractable even when concurrency is involved. In contrast, model checking of program code results in a much higher number of states to be checked, especially for large programs. If concurrency is added, model checking of such programs becomes intractable.

Theorem proving takes a different approach. First, models and verification properties can be defined in a more general logic, e.g. higher-order logic (HOL). This increases expressiveness and readability. Second, there is no need for modelled program state to be enumerable and the size of the state space is irrelevant. Furthermore, there are a variety of verification properties, such as functional correctness, that are very hard to express and prove with model checking and therefore require a theorem-proving approach. However, there is considerably less automation compared to model checking. Machine-assisted/checked theorem proving requires a proof engineer to provide creative input to guide the verification framework in finding a proof using deductive reasoning, which is a complex and time-consuming task. In case of concurrent-program verification, proof complexity increases sharply with the complexity of the synchronisation mechanisms used because the proof has to cover all conceptual scenarios that can arise from concurrent execution of the program code in question.

Examples of interactive theorem proving frameworks are *Isabelle/HOL* [NPW02] and *HOL4* [HOL]. There are also hybrid approaches such as automated first-order theorem proving. An example of such a verification framework is *VCC*, which we present in Section 3.3.1.

1.3 Problem

In the last section, we explained why formal verification becomes intractable very quickly as soon as concurrency is involved. For this reason, past kernel verification projects avoided having to reason about concurrency. As a result, the current state of research restricts computer systems that require a verified kernel to running on one CPU.

In order to make multiprocessor kernel verification tractable, we need to develop a suitable verification framework and choose the right level of abstraction (machine model) to work with. The choice of the verification property to be proved impacts tractability as well. In the end, however, tractability primarily depends on the verification target itself, i.e. the kernel. In-kernel synchronisation and communication between CPUs have to be designed with utmost care. There is a delicate trade-off: A simple design aids tractability but may adversely affect performance and scalability while elaborate scalable kernel designs most likely make verification intractable.

We aim to solve the problem of intractability by carefully crafting a suitable multiprocessor kernel design. While trying to maximise tractability, we also take into account the design's expected performance and scalability.

Designing the entire verification target (kernel) from scratch seems to be a promising approach. However, verifying a general-purpose kernel down to implementation level requires a large effort, even when no concurrency is involved. The verification effort can also include the development of the verification framework. For example, proving functional correctness of the seL4 microkernel required 20 person years (py) of which 11 py were kernel-specific efforts [KEH$^+$09]. The rest went into developing the verification framework, proof libraries etc. Aiming for a verified multiprocessor kernel, it is therefore desirable to leverage as much as possible from an existing kernel, its verification framework and proof.

1.4 Approach

In this thesis, we aim to demonstrate that it is possible to extend a verified uniprocessor kernel to utilise multiple CPUs and leverage the existing proofs to obtain a verified multiprocessor version of that kernel. We want to show that it requires relatively low effort, compared to the kernel's initial verification.

To this end, we introduce the *clustered multikernel*, a point in the design space of multiprocessor kernels. The main feature of this design is that it reduces concurrent data access to a minimum while offering a configurable trade-off between scalability and flexibility. This is possible by confining the required concurrent data accesses such that reasoning about them can be decoupled from reasoning about the kernel's functionality. Furthermore, the design eases conversion of a uniprocessor kernel into a multiprocessor kernel. For this purpose, we present a *conversion scheme* to convert a uniprocessor kernel into a clustered multikernel.

The process of reusing an existing proof in a new context by adapting or extending it is called *lifting*. For example, a theorem about a kernel-internal function can be reused in a multiprocessor context if we prove that no concurrency is introduced for that particular function.

To do this, we contribute a formal *refinement lifting framework*, which exploits the clustered multikernel's design features, specifically the confined concurrent data access. This allows it to lift most of the uniprocessor kernel's refinement proof such that it applies to the clustered-multikernel version (under certain assumptions). The support for handling the confined concurrency is added to the existing verification framework in a non-intrusive way. The refinement lifting framework accounts for weak memory ordering exhibited by total-store-order (TSO) multiprocessor architectures.

To demonstrate the practicability of the conversion scheme and the refinement lifting framework, we report on our experience with applying them to the seL4 microkernel [KEH+09].

All formal specifications and proofs presented in this thesis are machine-checked and typeset in the interactive theorem prover Isabelle/HOL [NPW02].

1.5 Contributions

The primary contribution of this thesis is an approach to retaining the formal guarantees of a verified uniprocessor kernel when converting it into a multiprocessor kernel. More specifically:

- We introduce the **clustered multikernel**, a point in the design space of multiprocessor kernels. This design reduces concurrent data access to a minimum while offering a configurable trade-off between scalability and flexibility. Furthermore, the design eases conversion of a uniprocessor kernel into a multiprocessor kernel.
- We present a **conversion scheme** to convert a uniprocessor kernel into a clustered multikernel.
- Our **refinement lifting framework** lifts the converted kernel's functional-correctness proof such that it applies to the clustered-multikernel version, under certain assumptions.
- We identify these assumptions and substantiate them; most of them formally, some of them informally.

- We report on our experience with applying the conversion scheme and the refinement lifting framework to the seL4 microkernel. One result of this process is **seL4::CMK**, a proof-of-concept clustered-multikernel implementation of seL4.
- We identify the limitations of our approach and elaborate on how they can be overcome in future work.

The work presented in this thesis is solely the author's work. Contributions by others are clearly identified as such.

1.6 Overview

The thesis is organised as follows: Chapter 2 summarises the background necessary to understand the remainder of this thesis. In Chapter 3, we present related work. Chapter 4 introduces the concept of the clustered multikernel, presents the conversion scheme and seL4::CMK. The main contribution of this thesis—the refinement lifting framework—is described in Chapter 5 and Chapter 6. Finally, we conclude and talk about future work in Chapter 7.

Chapter 2

Background

In this chapter, we summarise the background necessary to understand the remainder of this thesis. First, Section 2.1 introduces terminology and formal notation that will be used in definitions, lemmas and theorems. In Section 2.2, we present the refinement calculus that is the foundation of seL4's functional-correctness proof and of the refinement lifting framework contributed by this thesis. Section 2.3 introduces the concept of state monads, which are used for abstract specifications in seL4 and in this thesis.

In Section 2.4, we elaborate on OS kernel architectures (such as the *microkernel*), after which we present the seL4 microkernel in Section 2.5 and its formal verification in Section 2.6.

The last two sections cover general multiprocessor knowledge: Section 2.7 provides the necessary background about weak memory ordering, while Section 2.8 elaborates on the challenges of multiprocessor kernel programming, on common synchronisation mechanisms and multiprocessor design choices.

2.1 Formal Notation

This section gives an overview of the Isabelle/HOL [NPW02] notation used in this thesis.

Lemmas/Theorems In Isabelle, lemmas and theorems consist of a *goal* and zero or more *assumptions*. In the following example, P and Q are assumptions and $P \wedge Q$ is the goal. Valid notations are:

$$P \Longrightarrow Q \Longrightarrow P \wedge Q \qquad [\![P;\ Q]\!] \Longrightarrow P \wedge Q \qquad \frac{P \qquad Q}{P \wedge Q}$$

Types The notation $x :: t$ means that term x is of type t. Definitions of new types are started with the Isabelle keyword **types**.

Isabelle also supports *type variables*, which are prefixed with an apostrophe (e.g. $'a$). Type variables enable the definition of generic types which can later be instantiated with specific types.

Unit The type $unit$ has a single element $()$. It is normally used to instantiate type variables in case we do not require the instantiated type to carry any information. For example, we can use $unit$ as type for return values of functions that do not return anything.

Datatypes Datatypes are types that contain different kinds of content depending on a *constructor*. For example: The type $bool$ is a datatype with constructors True and False;

and natural numbers (`nat`) are defined as a recursive datatype with a constructor for the number `0` and a constructor for the *successor* of an existing `nat`:

```
datatype bool = True | False
datatype nat = 0 | Suc nat
```

Option Types An `'a option` is a datatype with two constructors:

```
datatype 'a option = ⌊'a⌋ | None
```

Hence, an `'a option` either contains an element of type `'a` or None. Note that `'a` is a type variable and is normally instantiated with a specific type when using the option type. For example, we can define a variable as `nat option`, which means that it either holds a natural number or None.

The function the can be used to extract element `x` from option `⌊x⌋`. Note that the None returns an undefined value.

Functions The type `a ⇒ r` denotes a function with a parameter of type `a` and returning a value of type `r`. Function application is written as `f x y`, which is equivalent to `(f x) y`.

Functions are usually defined using a single equation. However, it is also possible to use multiple equations, e.g. for defining recursive functions.

In order to instantiate function-parameter variables during function application, Isabelle uses *pattern matching*. This means that for functions that are defined with a single equation, the function parameters on the left hand side of the equation are usually variables that also appear on the right hand side. In this case, pattern matching is trivial. For functions defined with multiple equations, function parameters are normally (partially) instantiated in these equations. In such a case, Isabelle employs pattern matching in order to decide which equation is applicable for a particular function application. The *dummy* notation "_" can be used to match anything.

Constructing functions within terms is possible by using the lambda notation (e.g. `λx. x + 1`).

Isabelle functions are total. Partial functions are defined by using an option type as range. Either of the following syntax defines such a function:

```
'a ⇒ 'b option
'a ⇀ 'b
```

An empty partial function can be written as empty, which is defined as `λ_. None`. The domain of a partial function can be obtained by dom.

Isabelle supports *function updates*, which are written as `f(x := y)`. In this example, the function `f` is updated such that for an input of `x`, it returns `y`. For other inputs, the result values of `f` are unchanged. For partial functions, `f(x ↦ y)` can be used as an abbreviation for `f(x := Some y)`.

Pairs A pair with members `a` and `b` is denoted by `(a, b) :: 'a × 'b`. Its members can be accessed with the functions fst and snd, respectively.

Tuples Tuples are nested pairs, e.g. `(a, b, c, d) :: 'a × 'b × 'c × 'd` is internally represented as type `'a × ('b × ('c × 'd))`.

Sets A set is a function from the element type to `bool`, i.e. defining an `'a set` variable actually defines an `'a ⇒ bool` variable. The empty set is written as ∅ and the universal set

as UNIV. The image of a set `S` under function `f` is denoted by `f ` S`. The Cartesian product of two sets is written as `S × T`. The predicate finite tests whether a set is finite.

The syntax `{x..y}` can be used to specify a set that contains all members of an ordered type (e.g. `nat`) between and including `x` and `y`. The syntax `{x..<y}` returns the same set, except that `y` is not included.

Relations A relation is a set of pairs. The image of a set `S` under relation `R` is denoted by `R `` S`. The composition of two relations is written as $R_1$ `O` R_2.

Collections A collection is written as `{x | P x}` and is a set in which each element satisfies predicate `P`.

Lists Another example of a recursive datatype is the type `list`. It has a constructor for the empty list and a constructor for prepending an element to an existing list:

datatype `'a list = [] | 'a · 'a list`

Note that the constructors above are directly defined with their syntax: `[]` for the empty list and the dot notation for prepending to a list.

The syntax `[a, b, c]` creates a list containing the elements `a`, `b` and `c` in this order. The n'th element of a list can be directly obtained form a list `xs` with $xs_{[n]}$. Two lists `xs` and `ys` are concatenated by typing `xs @ ys`.

Isabelle provides several functions which facilitate working with lists:

hd	:: `'a list ⇒ 'a`
tl	:: `'a list ⇒ 'a list`
length	:: `'a list ⇒ nat`
set	:: `'a list ⇒ 'a ⇒ bool`
foldl	:: `('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b list ⇒ 'a`
map	:: `('a ⇒ 'b) ⇒ 'a list ⇒ 'b list`
filter	:: `('a ⇒ bool) ⇒ 'a list ⇒ 'a list`

The hd function returns the first element of a list while tl returns the remainder of the list. We can calculate the length of a list with length or use set to construct a set from the list's elements. We can reduce a list to one element by using foldl which repeatedly applies a function, and in each iteration, takes as arguments an element of the list and the result of the last iteration. The map function applies a function to each element of the list, while filter removes elements from a list which do not satisfy the predicate passed as first argument. Isabelle supports following syntax for filter:

`[x ← xs. P x]` = filter `P xs`

Records A record consists of named fields, each having an assigned type. Records can be initialised with the notation `(|field_A = x, field_B = y, field_C = z|)` etc. The term `field_A R` returns the value of `field_A` of record `R`. Changing the value of an existing record's field is possible with a *record field update*: The syntax `R(|field_A := x|)` returns a record equal to `R`, except that `field_A` contains the new value `x`.

Let Notation The syntax let `x = y` in `E` returns the expression `E` with every occurrence of `x` replaced with `y`.

Case Notation For case distinction on datatypes, we can use the following syntax:

```
case x of None ⇒ 0 | ⌊v⌋ ⇒ v
```

In this example, the type of `x` is the option type. If `x` is None, the expression returns `0`. If `x` is `⌊v⌋`, the expression returns `v`. Pattern matching in the conditions is possible. The pattern "_" can be used to match any datatype constructor.

2.2 Refinement Calculus

Functional correctness of seL4 is proved via *data refinement* [dRE98]. This refinement calculus is formalised in Isabelle as described in Cock et al. [CKS08]. In this section, we introduce the formalisation on the level of detail necessary to understand seL4's formal verification (Section 2.6) and the refinement lifting framework (Chapter 5 and Chapter 6).

On each level of abstraction in a refinement proof, the program in question is modelled as an *abstract data type (ADT)*, which consists of 3 functions operating on 3 basic types.

Definition 2.2.1 (Abstract Data Type)

```
record ('priv, 'obs, 'j) adt_t =  Init :: 'priv ⇒ bool
                                  Step :: 'j ⇒ 'priv × 'priv ⇒ bool
                                  Fin  :: 'priv ⇒ 'obs
```

The *observable state* type `'obs` needs to be the same on each abstraction level. It is the common representation of the program state that links all abstraction levels. The *private state* type `'priv` is supposed to be different on each abstraction level. On lower levels, it tends to be more detailed than on higher levels. The type instantiated for `'j` holds all *transitions* of the system, i.e. each member of this type represents one system transition. Thus, `'j` is normally instantiated with a datatype, so each constructor represents a named transition.

The *initialisation function* Init returns a set of initial private states `'priv`, which allows us to non-deterministically model how the program is bootstrapped.[1] The Step function models the behaviour of the program while running. For each *transition* of type `'j`, it returns the set of allowed *steps*, i.e. a relation between old and new private states. Finally, Fin projects the private state space into the observable one.

Execution of a program (represented as an ADT) is defined as follows.

Definition 2.2.2 (Execution)

```
steps :: ('j ⇒ 'priv × 'priv ⇒ bool) ⇒ ('priv ⇒ bool) ⇒ 'j list ⇒ 'priv ⇒ bool
steps δ ≡ foldl (λS j. δ j `` S)

execution :: ('priv, 'obs, 'j) adt_t ⇒ 'j list ⇒ 'obs ⇒ bool
execution A js ≡ Fin A ` steps (Step A) (Init A) js
```

The initialisation function Init `A` returns a set of private states `S`. For each transition `j` in the list `js`, the set of states `S` is transformed by applying the relation returned by Step `A j`. Finally, Fin `A` projects the set of resulting private states into observable ones.

Refinement between an *abstract* program (ADT `A`) and a *concrete* program (ADT `C`) is defined as follows.

[1] Note that the original formalisation [CKS08] is slightly more general: The Init function takes an observable state as parameter. This would enable an Init function to return different sets of initial private states depending on this parameter. However, as seL4 does not make use of this, it ignores the parameter. Therefore, we modified the formalisation slightly by dropping the parameter. Doing so had no impact on seL4's refinement statement and proof because we only removed a parameter that had been ignored before.

Definition 2.2.3 (Refinement)

```
op ⊑ :: ('priv_C, 'obs, 'j) adt_t ⇒ ('priv_A, 'obs, 'j) adt_t ⇒ bool
C ⊑ A ≡ ∀ js. execution C js ⊆ execution A js
```

ADT C refines ADT A if, for all transition lists js, the resulting set of observable states of executing ADT C is a subset of executing ADT A.

Note that an important aspect of refinement is that it is transitive.

Theorem 2.2.4 (Transitive Refinement)

```
⟦C ⊑ B; B ⊑ A⟧ ⟹ C ⊑ A
```

Refinement is commonly proved via *forward simulation* [dRE98], which is defined as follows.

Definition 2.2.5 (Forward Simulation)

```
fw_sim :: ('priv_A, 'obs, 'j) adt_t
        ⇒ ('priv_C, 'obs, 'j) adt_t ⇒ ('priv_A × 'priv_C ⇒ bool) ⇒ bool
fw_sim A C R ≡ Init C ⊆ R `` Init A ∧
               (∀ j. R O Step C j ⊆ Step A j O R) ∧
               (∀ s s'. (s, s') ∈ R ⟶ Fin C s' = Fin A s)

op ⊑F :: ('priv_A, 'obs, 'j) adt_t ⇒ ('priv_C, 'obs, 'j) adt_t ⇒ bool
C ⊑F A ≡ ∃R. fw_sim A C R
```

In order to show that ADT C forward simulates ADT A, we have to define a *refinement relation* R which relates their private state spaces to each other. We must prove that the relation is established by Init and preserved by Step for all transitions j, and that Fin projects related private states into equal observable states. We call these proofs *correspondence proofs*.

Theorem 2.2.6 *Forward simulation implies refinement.*

```
C ⊑F A ⟹ C ⊑ A
```

In correspondence proofs, it is often essential that we can rely on invariants restricting the set of private states we have to cover in those proofs.

Definition 2.2.7 (Invariance)

```
op ⊨ :: ('priv, 'obs, 'j) adt_t ⇒ ('priv ⇒ bool) ⇒ bool
A ⊨ I ≡ Init A ⊆ I ∧ (∀ j. Step A j `` I ⊆ I)
```

The invariants I hold throughout execution of ADT A if they are established by Init and preserved by Step for all transitions j. We call these proofs *invariant proofs*.

Definition 2.2.8 (Forward Simulation with Invariants)

```
fw_sim_inv :: ('priv_A, 'obs, 'j) adt_t
            ⇒ ('priv_C, 'obs, 'j) adt_t
               ⇒ ('priv_A × 'priv_C ⇒ bool)
                  ⇒ ('priv_A × 'priv_C ⇒ bool) ⇒ bool
fw_sim_inv A C R I ≡ Init C ⊆ R `` Init A ∧
                     (∀ j. (R ∩ I) O Step C j ⊆ Step A j O R) ∧
                     (∀ s s'. (s, s') ∈ R ∩ I ⟶ Fin C s' = Fin A s)
```

The definition is identical to Definition 2.2.5 of fw_sim, except that we restrict the set of private states according to the invariants I for the Step and Fin clauses. It states that forward-simulation holds under the assumption that the invariants hold.

Theorem 2.2.9 *Forward simulation with invariants implies forward simulation if the invariants hold in both ADTs.*

$$[\![A \models I_a;\ C \models I_c;\ \mathsf{fw\_sim\_inv}\ A\ C\ R\ (I_a \times I_c)]\!] \Longrightarrow C \sqsubseteq_F A$$

2.3 Hoare Logic on State Monads

Recall from Section 2.2 that on each abstraction level of a refinement proof, the behaviour of the program in question is defined by the ADT's Init and Step functions. These functions define the program's behaviour by modifying the ADT's private state.

The goal of a specification is to model the behaviour of a program in a way that is more abstract than the implementation and therefore easier for humans to write, read and reason about. As such, it is desirable to have a way of specifying these functions that facilitates classical program reasoning. A very popular way of reasoning about program code is by using Hoare triples. Therefore, we want to be able to use them to reason about how a specific function modifies the ADT's private state. To this end, seL4 uses *non-deterministic state monads* for its specifications.

In programming, *state monads* provide a way of using an imperative programming style in a functional programming language. A state monad facilitates accessing a global program state from within the program's functions. Such *monadic functions* consist of a sequence of calls to other monadic functions with each of them accessing the monad state sequentially.

State monads are useful for program specifications for multiple reasons: For example, their model of computation is purely functional. As such, they can be formalised in higher-order logic (HOL) (and therefore in Isabelle/HOL) without assumptions or axiomatisations. Furthermore, the notion of a global program state enables reasoning about this state at the beginning and end of a monadic function, e.g. we can define Hoare triples whose pre- and postconditions talk about the program state.

In functional programming, state monads are usually deterministic because the programs are intended to be executed. In specifications, however, non-deterministic state monads have several advantages over deterministic ones (as we will see below).

In this section, we present the formalisation of Hoare logic on non-deterministic state monads, originally described in Cock et al. [CKS08]. This formalisation is used in the specifications and proofs of seL4. We focus on the information necessary to understand seL4's formal verification (Section 2.6) and the refinement lifting framework (Chapter 5 and Chapter 6).

2.3.1 Non-Deterministic State Monads

The type of a non-deterministic state monad is defined as follows.

Definition 2.3.1 (Non-Deterministic State Monad Type)

```
types ('s, 'r) nondet_monad = 's ⇒ ('r × 's) set × bool
```

A state monad takes the monad state (type `'s`) as sole argument and returns a pair. The right side of the pair (snd) is the *failure flag*[2] (type `bool`). The left side of the pair (fst) is a non-deterministic set of possible *results* of the monadic computation. A result is a pair of

[2]The failure flag can be used to signal program failure. For example, the monadic library function fail always fails, while assert fails if a certain condition on the monad state is not met.

the return value of the function (type *'r*) and the monad state at the end of the function (type *'s*).

As an example, a monadic function foo that operates on a monad state of type *state* and returns a natural number (type *nat*) is declared as:

```
foo :: (state, nat) nondet_monad
```

The *monadic computation* (or *monadic execution*) is defined by the following two fundamental monad functions, called *constructors*.

The bind constructor defines how two monadic functions *f* and *g* are put in sequence.

Definition 2.3.2 (Bind Constructor)

```
bind :: ('s, 'r1) nondet_monad ⇒ ('r1 ⇒ ('s, 'r2) nondet_monad) ⇒
        ('s, 'r2) nondet_monad
bind f g ≡ λs. (⋃fst ` (λ(x, y). g x y) ` fst (f s),
                True ∈ snd ` (λ(x, y). g x y) ` fst (f s) ∨ snd (f s))
```

The overall result is calculated by taking the set of all possible results of *f*, feeding each of them into *g* and calculating the union of *g*'s results. The failure flag is set if it was set in either *f* or *g*.

The second constructor is the return function.

Definition 2.3.3 (Return Constructor)

```
return :: 'r ⇒ ('s, 'r) nondet_monad
return r ≡ λs. ({(r, s)}, False)
```

It returns the specified value *r* without changing the monad state and without failing. To facilitate reading and writing of the monad state, the following accessor functions are provided.

Definition 2.3.4 (Accessor Functions)

```
get :: ('s, 's) nondet_monad
get ≡ λs. ({(s, s)}, False)

gets :: ('s ⇒ 'r) ⇒ ('s, 'r) nondet_monad
gets f ≡ do s ← get;      return (f s) od

put :: 's ⇒ ('s, unit) nondet_monad
put s ≡ λ_. ({((), s)}, False)

modify :: ('s ⇒ 's) ⇒ ('s, unit) nondet_monad
modify f ≡ do s ← get;      put (f s) od
```

The function get returns the current monad state as a proper return value whereas gets applies a transformation function *f* before returning. This is useful, for example, if the monad state is a record and we only want to return one specific field. Neither get nor gets modify the monad state itself.

The function put sets the state given as argument as the new monad state, while modify is a combination of get and put. As parameter, it takes a simple state-modifier function *f*. Note that the return type *unit* models that a function has no return value.

None of the accessor functions presented above can fail (set the failure flag).

Do Notation

The gets and modify functions above are defined using the *do notation*. This notation is primarily a syntax for a recursive application of the bind constructor, which allows putting multiple monadic functions in sequence.

A *program block* is defined by an opening `do` and closing `od` token. In between, we can place calls to multiple monadic functions, separated by semicolons. Each semicolon represents the application of the bind constructor with its parameter `f` instantiated with the function before the semicolon and `g` instantiated with the remaining program after the semicolon. Thus, a program block with multiple semicolons represents a nested application of the bind constructor.

In order to handle return values, it is possible to prepend a left arrow to the monadic function call, which results in the return value of the function being available as local bound variable in the remainder of the program block. In the definition of modify, for example, the variable `s` contains the value returned by get and is available for put later. Internally, this variable is the first parameter of bind's function parameter `g`.

Non-Deterministic Choice

All monadic functions presented so far are deterministic. To help specifying a non-deterministic program, the following two monadic functions are provided.

Definition 2.3.5 (Non-Deterministic Alternative Execution)

```
alternative :: ('s, 'r) nondet_monad ⇒ ('s, 'r) nondet_monad ⇒ ('s, 'r) nondet_monad
alternative f g ≡ λs. (fst (f s) ∪ fst (g s), snd (f s) ∨ snd (g s))
```

This specifies that either monadic function `f` or `g` is executed. The failure flag is set if it is set by either `f` or `g`. The syntax `f ⊓ g` is an abbreviation for alternative `f g`.

Non-deterministic return values can be generated with the following non-deterministic variant of return.

Definition 2.3.6 (Non-Deterministic Selection)

```
select :: 'r set ⇒ ('s, 'r) nondet_monad
select A ≡ λs. (A × {s}, False)
```

The select function therefore returns any value that is a member of set `A`. The monad state is not changed and the failure flag is not set. Refer to Definition 5.4.23 for an example how select can be used.

2.3.2 Hoare Logic

A Hoare triple over a monadic function `f` is defined as follows.

Definition 2.3.7 (Hoare Triple)

```
{|P|} f {|Q|} ≡ ∀s. P s ⟶ (∀(r, s')∈fst (f s). Q r s')
```

`P` is a unary predicate over the monad state `s` while `Q` is a binary predicate over the return value `r` of `f` and the monad state `s'`. The Hoare triple states that if `P` holds for the monad state before executing `f`, then `Q` holds for the return value of `f` and the monad state after having executed `f`.

To facilitate reasoning about Hoare triples, a collection of basic rules (theorems) are defined for the constructors and accessor functions presented above.

Theorem 2.3.8 (Basic Rules)

$\{\!|P\ x|\!\}$ return x $\{\!|P|\!\}$
$\{\!|\lambda s.\ \forall x{\in}S.\ P\ x\ s|\!\}$ select S $\{\!|P|\!\}$

$\{\!|\lambda s.\ P\ s\ s|\!\}$ get $\{\!|P|\!\}$
$\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$ gets f $\{\!|P|\!\}$
$\{\!|\lambda s.\ P\ ()\ x|\!\}$ put x $\{\!|P|\!\}$
$\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}$ modify f $\{\!|P|\!\}$

$$\frac{\{\!|A|\!\}\ f\ \{\!|B|\!\} \qquad \forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}}{\{\!|A|\!\}\ \text{bind}\ f\ g\ \{\!|C|\!\}}$$

$$\frac{\{\!|P|\!\}\ f\ \{\!|Q|\!\} \qquad \{\!|P'|\!\}\ f'\ \{\!|Q|\!\}}{\{\!|\lambda x.\ P\ x \wedge P'\ x|\!\}\ \text{alternative}\ f\ f'\ \{\!|Q|\!\}}$$

In order to be able to apply these rules in a Hoare-triple proof, we need a way of weakening preconditions.

Theorem 2.3.9 (Weaken Precondition)

$$\frac{\{\!|Q|\!\}\ f\ \{\!|R|\!\} \qquad \forall s.\ P\ s \longrightarrow Q\ s}{\{\!|P|\!\}\ f\ \{\!|R|\!\}}$$

Instead of proving the Hoare triple with the precondition P, we can prove it with precondition Q if we prove that Q is weaker than P, i.e. P implies Q.

We now explain how to use the above rules to prove Hoare triples, with the example of proving the *weakest-precondition* rule for modify (which we presented before):

$\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}$ modify f $\{\!|P|\!\}$

In order to prove this Hoare triple, we first unfold the definition of modify. This results in the following proof goal:

$\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}$ $do\ s \leftarrow$ get; put $(f\ s)$ od $\{\!|P|\!\}$

If the unfolded function contains calls to functions we do not have rules for, we also unfold those functions' definitions. Alternatively, we define rules for those functions first.

Now we weaken the precondition by applying Theorem 2.3.9. This results in the following two subgoals to be proved:

$\{\!|?Q|\!\}$ $do\ s \leftarrow$ get; put $(f\ s)$ od $\{\!|P|\!\}$
$P\ ()\ (f\ s) \Longrightarrow ?Q\ s$

Note that the precondition has been replaced with the schematic variable $?Q$. In Isabelle, schematic variables can be instantiated at any point in the proof. Normally, it is done by pattern matching when applying a rule.

We apply the bind rule to the first subgoal because bind is the outermost function in that Hoare triple (represented by the do notation). Applying this rule results in the first subgoal to be replaced with two new subgoals, one each for get and put. Applying their respective rules removes those subgoals and, at the same time, instantiates $?Q$ with $\lambda s.\ P\ ()\ (f\ s)$. This results in the following last subgoal:

$P\ ()\ (f\ s) \Longrightarrow P\ ()\ (f\ s)$

This a pure HOL formula without monad or Hoare-logic formalism. In our example, the final goal is trivial because the precondition of the Hoare triple we proved is in fact the weakest precondition. In other examples, this last goal can be more complex to prove.

In order to reduce manual proof work, the verification framework provides a *verification condition generator* (VCG). The VCG automates the steps described above, i.e. it automatically weakens preconditions and applies rules from a predefined rule set. Newly proved rules can be added to this set, which progressively extends the automation to higher levels of abstraction.

2.4 OS Kernel Architectures

The purpose of this section is to give an introduction to common OS kernel architectures for readers not familiar with this topic.

By definition, an OS *kernel* comprises all OS code that runs in the CPU's privileged mode. All other OS code and the application code run at *user level*.

The basic architecture of an OS kernel generally falls into one of the following categories: (1) *microkernel*, or (2) *monolithic kernel*.

2.4.1 Microkernels

In a microkernel-based OS, the kernel only implements OS functionality that *requires* running in the CPU's privileged mode. This usually comprises context switching, memory management and bottom-level interrupt handling. All other OS services are implemented as *servers* at user level. Typically, each server is implemented as a process, which runs in its own virtual address space. Examples of such servers are device drivers, file systems, network stacks, naming services and high-level process management. Hence, a microkernel-based OS is sometimes called *multiserver OS* [GJP+00].

All requests for OS services and all coordination between OS services requires *interprocess communication (IPC)*, which is provided by the microkernel. A simple OS service call requires two IPC transfers: one for the request and one for the response. Each IPC transfer requires two *CPU mode switches* (in and out of the kernel), and a *context switch* from the sender's address space to the receiver's address space. These are time-consuming operations on most CPU architectures. As a consequence, IPC is the most critical performance bottleneck in microkernel-based OSes [Lie93].

Advantages

In a microkernel-based OS, services run isolated from each other. If they have to communicate with each other, APIs and communication channels have to be explicitly defined. This requires well-defined interfaces between OS services, which reduces bug density and facilitates formal analysis.

The isolation also increases robustness of the OS because faults are contained within an OS service. A misbehaving service cannot crash the kernel or other services, it can only crash itself. It is possible to implement a *monitor* to detect this and restart crashed services [Her10]. For example, if the network device driver crashes, it can be restarted within a few milliseconds. While this might result in the loss of a few network packets, such an event will most likely not be noticed at application level.

A microkernel allows a minimal *trusted computing base (TCB)*. Remember that a system's TCB comprises all code that has to be trusted in order for a given security property to hold. Code outside the TCB cannot violate the security property. The OS kernel is always in the TCB because it runs in the CPU's privileged mode. However, OS services are only in the TCB if they are relevant for the security property in question. Microkernels therefore

facilitate OS designs that provide minimal TCBs for different security properties.

Drawbacks

The obvious problem with microkernel-based OSes is that they heavily use IPC, which is an inherent performance bottleneck. While it is possible to reduce IPC latency drastically with careful microkernel design and implementation [Lie93], the overhead of the numerous CPU mode switches and context switches will always be there.

2.4.2 Monolithic Kernels

In contrast to microkernels, monolithic kernels run most OS services within the kernel, i.e. in the CPU's privileged mode. Therefore, coordination between OS services does not require any CPU mode switches or context switches. User-level processes are mainly used to run applications, which request OS services from the kernel directly via *system calls*. A system call only requires two CPU mode switches (in and out of the kernel) but no context switch. IPC in monolithic kernels is mainly used for communication between applications and is therefore a lot less critical to overall system performance than for microkernels.

The most widely used kernels nowadays (e.g. Windows, Linux, OS X, Unix) are monolithic.

Advantages

Monolithic kernels have a performance advantage over microkernels because they require fewer CPU mode switches and context switches to complete the same tasks. There is also more freedom in how OS services can interact with each other. The possibilities range from synchronous models such as direct function calls or variable accesses across OS services to asynchronous models in which OS services communicate via explicit messages. The latter model is similar to the one used in a microkernel-based OS. However, the communication can be implemented more efficiently because no CPU mode switches or context switches are necessary.

Drawbacks

Monolithic kernels are generally less robust than microkernels. If an OS service misbehaves, it can crash the entire system because it is not isolated from other OS services. In practice, the most problematic OS services are device drivers. Chou et al. [CYC+01] reports a three to seven times higher bug density in device-driver code than other system components. In a monolithic system, each of these bugs has a high potential of crashing the entire system.

From a security perspective, monolithic kernels have the drawback that the OS services within the kernel are always in the trusted computing base (TCB), even the ones that would not be relevant for the security property in question.

2.4.3 Hybrid Kernels

There are kernels that do not strictly fall into one of the above categories. Such kernels are sometimes called *hybrid kernels*. They are a compromise between microkernels and monolithic kernels with some OS services implemented in the kernel and others implemented at user level. Normally, all performance-critical services are implemented in the kernel

(e.g. device drivers, timers) while the remaining services are implemented at user level (e.g. file systems, naming services).

It is also hard to draw an exact line between microkernels and hybrid kernels, or between hybrid kernels and monolithic kernels; and kernels sometimes cross these lines during their lifetimes. For example, *Mach* [ABB+86] started as a microkernel. In order to improve performance, more and more OS services were moved into the kernel. Over time, Mach became a hybrid kernel. Similarly, the Windows NT kernel started as a hybrid kernel. However, more and more OS services were moved into the kernel over time, mostly for performance reasons. Nowadays, the Windows kernel is considered a monolithic kernel.

2.5 The seL4 Microkernel

In Chapter 5 and Chapter 6, we apply the refinement lifting framework to the seL4 microkernel. The current section provides the necessary background to readers not familiar with seL4 and its internals.

seL4 [EDE08] is a third-generation microkernel, loosely based on L4 [Lie95]. It provides virtual address spaces, threads, and inter-process communication (IPC). The access-control mechanism—influenced by EROS [SSF99], KeyKOS [Har85] and CAP [NW77]—is based on capabilities, which grant access to kernel objects. Its performance is comparable to previous high-performance L4 microkernels.

Initially, seL4 was implemented for ARMv6, which is the version that is formally verified. Today, unverified implementations exist for other ARM versions, for x86 and for x64.[3] There are plans to formally verify the x86 version, of which an executable intermediate specification has already been written in Haskell.

The verified version of seL4 comprises 8700 lines of C code and 600 lines of assembly code [KEH+09].

2.5.1 Kernel Objects

Every type of dynamic kernel data structure is represented by a *kernel object* type. The kernel does not implicitly allocate memory for kernel objects. Instead, they need to be explicitly created from user level. For such an operation, the user needs to provide *untyped memory*, which the kernel then uses to store the kernel object's state. Untyped memory cannot be accessed directly by the user.

2.5.2 Capabilities

A *capability* is a protected reference to a kernel object. A kernel object can only be used by a thread which possesses a capability to it. Most kernel object types have *methods* which operate on the object's state. They are executed by *invoking* a capability that references the kernel object in question.

For example, the user can invoke the *retype* method of an untyped-memory capability in order to create kernel objects within the untyped memory region covered by the capability.

Capabilities are stored in *CNodes*, which are kernel objects themselves.

Some kernel object types only have methods but no state. Their purpose is to provide methods that provide kernel functionality which is not directly associated with specific

[3] Throughout this thesis, we use the term *x86* for the *IA-32* and *AMD32* architectures, and the term *x64* for the *Intel 64* and *AMD64* architectures.

kernel state. Capabilities of such object types do not reference object instances (there are none). Instead, they convey the right to call the methods associated with the particular kernel object type.

2.5.3 Virtual Address Spaces

The seL4 kernel does not define the structure of virtual address spaces, i.e. virtual-address mappings are managed at user level. For example, a page fault triggers an exception message to be sent to the designated user-level *pager* thread of the address space. The pager implements the policy of how the page fault should be handled, e.g. which physical frame the page should be mapped to.

User-level frames are the only type of memory that can be directly accessed from user level. They are created from untyped memory at the user's request. Before they can be accessed, they need to be mapped into a virtual address space. Address spaces are constructed by creating and linking the necessary page-table objects. An address space is identified by a capability to its top-level page table.

2.5.4 Threads

Threads are scheduled by the kernel. After a thread has been created, it has to be assigned to an address space in order to be useful. It is possible to change the assigned address space during the lifetime of the thread. An address space can have zero, one or multiple threads assigned to it. There is no notion of a primary/main thread in an address space, neither is there a notion of a *process* or *task*. The state of a thread is stored in a *thread control block* (TCB) object.

There are no "kernel threads", i.e. there are no threads that execute entirely in the kernel (except for the idle thread). All system calls are executed in the context of the thread that triggered them at user level.

2.5.5 Inter-Process Communication (IPC)

IPC between threads takes place via *endpoints*. An endpoint is an independent kernel object. Threads in possession of an endpoint capability can invoke it to send or receive IPC messages via that endpoint. As soon as an endpoint has been invoked with both a send and a receive request, an IPC transfer is started between the two invoking threads. Hence, IPC can only take place between threads that possess capabilities to the same endpoint. The semantics of IPC are the same regardless of whether the two communicating threads are in the same address space or not.

There are synchronous and asynchronous IPC endpoints. When sending to or receiving from a synchronous endpoint, a thread is blocked until a send/receive match is found, i.e. until the IPC transfer takes place. Synchronous endpoints do not store IPC messages. Messages have a maximum payload size of 480 bytes (120 message words of 32 bits).

Sending to asynchronous endpoints is always non-blocking. The message payload is a single 32-bit word, which is stored in the endpoint. The endpoint cannot store multiple messages. If multiple send invocations are performed between receive invocations, the messages are combined to one single message by performing a bitwise OR operation.

2.5.6 Device Drivers and IRQs

As is common in microkernel-based systems, device drivers are implemented outside the kernel. In order to access device registers, the user-level driver is provided with *device frames*. These are special kernel-created frames that cover a device's memory-mapped registers.

IRQs are delivered to asynchronous endpoints to which a device driver listens. IRQ kernel objects allow the user to configure which IRQ is delivered to which endpoint. They are also used by the device driver to enable, disable or acknowledge an IRQ.

2.5.7 Preemption

seL4 cannot be directly interrupted while handling an event. Interrupts are turned off during kernel execution. In order to avoid high interrupt latency, seL4 has explicit preemption points in long-running operations. Between each iteration of these operations, seL4 checks for pending IRQs, and if any are detected, triggers a preemption exception. The exception is propagated up to the main kernel entry function where we leave the kernel and return to user level. Only here, the IRQ is triggered and we enter the kernel again to handle the IRQ. After having handled the IRQ, the kernel returns to user level, where the original system call is restarted and the long-running operation continued.

2.6 Formal Verification of seL4

In Chapter 5 and Chapter 6, we apply the refinement lifting framework to the seL4 microkernel. These chapters build on the details about seL4's formal verification presented in this section.

Formal verification of seL4 started with the *L4.verified* project [KEH+09], which proved that the C implementation of seL4 refines seL4's abstract specification. The main advantage of having a refinement proof is that it *transfers* so called *safety hyperproperties* [CS08] proved on the abstract level down to the concrete level(s). This fact was leveraged by a follow-on project [SWG+11], which proved that seL4 enforces authority confinement and integrity, and based on this, that it enforces noninterference[4] [MMB+12] and information flow [MMB+13]. Each of these conditions is a safety hyperproperty and was proved on the abstract level. Because of the refinement proof, we know that they also hold on the implementation level.

In general, reasoning about an abstract specification requires an order of magnitude less effort than reasoning directly about the implementation [KMG+11].

The verification property this thesis focuses on is refinement. Therefore, this section summarises the basics of seL4's refinement proof, originally published in Cock et al. [CKS08] and Klein et al. [KSW10]. We concentrate on the specific details that will be relevant for the application of the refinement lifting framework to seL4 in Chapter 5 and Chapter 6.

2.6.1 Assumptions

The L4.verified project proved functional correctness of the C code. This means that everything below the semantic level of C is assumed to be correct. This includes the hardware,

[4] Note that noninterference was proved on a variant of seL4's abstract specification from which nondeterminism had been removed.

C compiler/linker and 600 lines of assembly code, which implements functionality that cannot be written in C.

In the meantime, a follow-on project [SMK13] has removed the compiler assumption. The project proved refinement between the assembly output generated by the compiler and the C implementation.

A further assumption of L4.verified is that seL4 correctly operates hardware such as the TLB (translation-lookaside buffer), CPU caches and interrupt controller. This assumption exists because seL4 models the underlying hardware in an abstract way with hardware operations modelled as *abstract machine operations*. Consider the following example: After unmapping a user-level frame from a virtual address space, the kernel has to invalidate any TLB entry that matches the frame's former virtual address. If the TLB-invalidation operation is missing, e.g. because of a kernel bug, the frame can still be accessed via its old virtual address, even though it has been unmapped. The L4.verified hardware model is not detailed enough to capture such a situation. Access to a user-level frame is impossible in the model as soon as the frame has been unmapped, no matter how the TLB is operated.

Nonetheless, the L4.verified project proved that the implementation performs a certain hardware operation whenever the abstract specification performs it. This gives a high assurance that the hardware is operated correctly in the implementation, but only under the assumption that it is correctly operated in the abstract specification. For example, if a TLB-invalidation operation is missing in the implementation and the specifications, the proof succeeds even though the kernel is faulty. In case of seL4, correctness of such operations is validated by testing. It would also be possible to extend the machine model with enough detail to allow a proof about correct use of machine operations.

Finally, it is assumed that seL4's boot code is correct. The boot code comprises 1200 LOC of the total 8700 LOC. Formally, this means that for both refinement steps, correspondence proofs and invariant-establishment proofs are axiomatised for the Init functions of all ADTs. Note however, that there is nothing fundamental preventing such a proof.

For the applicability of the refinement lifting framework contributed in Chapter 5 and Chapter 6, it does not matter whether the correctness of the boot code is axiomatised or proved.

2.6.2 Refinement Steps

Refinement of seL4 is proved in two steps (Figure 2.1). The first step proves refinement between the *abstract specification* and the *intermediate specification*. These are specified in monadic style using the framework described in Section 2.3. The intermediate specification is automatically derived from a prototype implementation written in Haskell.

The second step proves refinement between the intermediate specification and the C implementation. The C code is directly parsed into an Isabelle-internal representation that is based on SIMPL [Sch06].

According to Theorem 2.2.4, this implies overall refinement, i.e. between the abstract specification and the C implementation. Both steps use the refinement calculus described in Section 2.2.

In this thesis, we concentrate on the first refinement step. However, our refinement lifting framework is applicable to the second step as well. Where necessary, we point out differences.

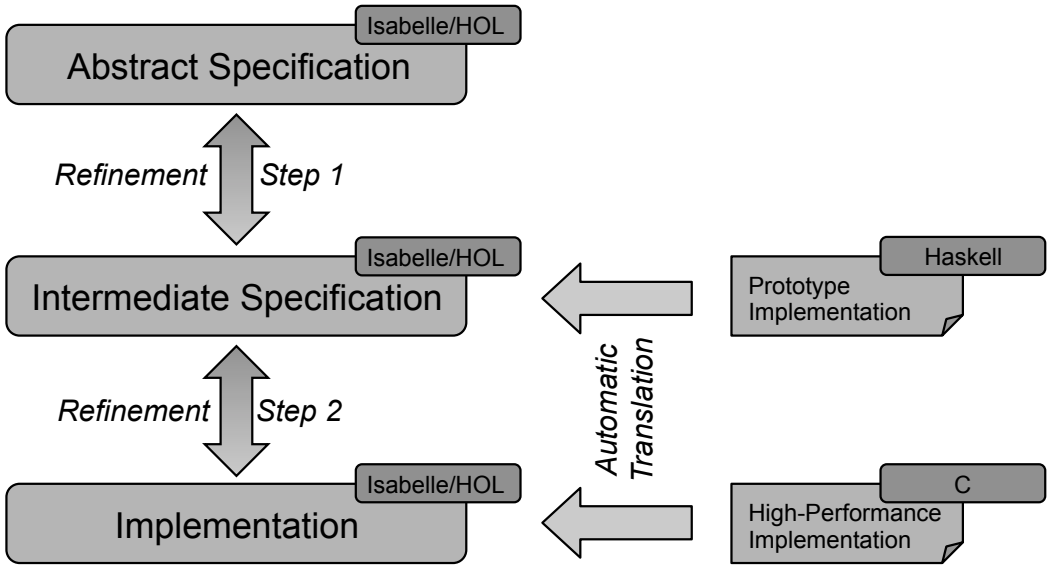

Figure 2.1: *L4.verified Overview*

2.6.3 Global Automaton

In this section, we describe how the refinement calculus of Section 2.2 is applied to seL4. Applying it means defining how the basic types and functions of the involved ADTs are instantiated. The resulting instantiated types will be used when applying the refinement lifting framework to seL4.

The private states of seL4's ADTs have the following type.

Definition 2.6.1 (Private State Type)

types *'s global_state = (user_context × 's) × mode × event option*

This type is parameterised with the type variable *'s*, which has to be instantiated with the actual type of the private state on a particular abstraction level. For example, the type of the private state on the abstract level is *state global_state* whereas for the intermediate level, it is *kernel_state global_state*.

This means that the remainder of the tuple is the same for both abstraction levels: The type *user_context* contains the values stored in the CPU's registers. The *event option* is an option of a datatype that captures whether a system call, exception or IRQ happened.

The datatype *mode* models in which mode the CPU currently operates.

Definition 2.6.2 (Mode Type)

datatype *mode* = UserMode | KernelMode | IdleMode

We define the following four transitions.

Definition 2.6.3 (Transitions)

datatype *global_transition* = KernelTransition
| UserTransition
| UserEventTransition
| IdleEventTransition

The ADTs' observable state `observable` is simply defined as the abstract private state `state global_state`.

This concludes the instantiation of both ADTs' types. We are now ready to define the ADTs' functions Init, Step and Fin.

Definition 2.6.4 (Abstract Init Function)

```
Init_A :: node_id_t ⇒ state global_state ⇒ bool
Init_A ≡ {((empty_context, init_A_st), UserMode, None)}
```

We start running in UserMode with None events pending and an empty_context for the CPU registers. The initial private state after bootstrapping is init_A_st, i.e. this function represents the abstract specification of kernel bootstrapping. However, since bootstrapping is assumed to be correct, init_A_st has no definition. The necessary properties of this function are axiomatised.

The Init function of the intermediate specification is defined in a similar way. Kernel bootstrapping is axiomatised as well.

The Fin function on the abstract level is the identity function (id) because the observable state has the same type as the abstract private state. On the intermediate level, Fin is a projection of the intermediate private state (`kernel_state global_state`) to the abstract private state (`state global_state`) and therefore to the observable state (`observable`). The projection is defined according to the *refinement relation*, which defines how the abstract and intermediate private states are related.

The Step function of an ADT captures the behaviour of the kernel after it has been bootstrapped. On each abstraction level, Step combines the ADT's transitions with the CPU modes (defined in the private state) to implement a *global automaton* as depicted in Figure 2.2.

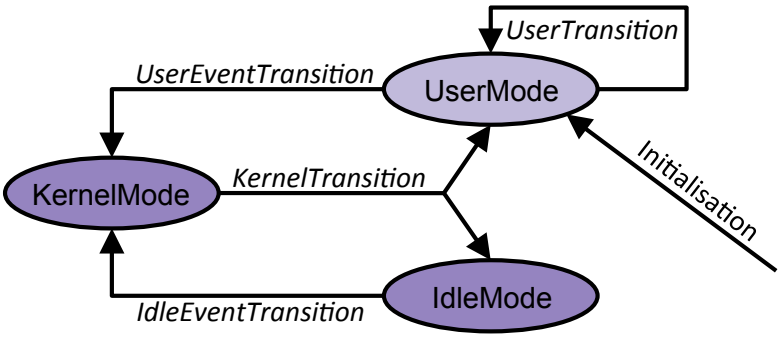

Figure 2.2: *Global Automaton of seL4*

Definition 2.6.5 (Global Automaton)

```
global_automaton do_user_op kernel_call t ≡
  case t of
  KernelTransition ⇒
    {((s, KernelMode, ⌊e⌋), s', m, None) | (s, m, s') ∈ kernel_call e}
  | UserTransition ⇒
      {((s, UserMode, None), s', UserMode,
        None) | (s, s')
                ∈ do_user_op
                    (λ(tc, um).
                        {(tc', um') | ∃v. ∃p∈dom um. um' = um(p ↦ v)})}
  | UserEventTransition ⇒ {((s, UserMode, None), s, KernelMode, ⌊e⌋) | True}
  | IdleEventTransition ⇒
      {((s, IdleMode, None), s, KernelMode, ⌊Interrupt⌋) | True}
```

This automaton is used to define the Step function of each abstraction level's ADT. It is parameterised with an operation for the UserTransition (*do_user_op*) and an operation for the KernelTransition (*kernel_call*). These operations are specific to a given abstraction level.

The automaton works as follows: After initialisation, the system is in UserMode, where it performs an arbitrary number of UserTransitions. In each such transition, it arbitrarily modifies the CPU's registers *tc* (type *user_context*) and one byte of *underlying memory* (*um*), which is modelled in *do_user_op*. The only way to exit UserMode is with a UserEventTransition, which sets an arbitrary event *e* and switches to KernelMode. Here, the event is handled by the kernel in a KernelTransition. This transition comprises the entire kernel functionality. At the end of the kernel call, the system either switches to UserMode or IdleMode, depending on what the scheduler decided in the KernelTransition. In IdleMode, the only thing that can happen is an Interrupt event, which takes the system back to KernelMode.

We are now able to define the ADTs of both levels, abstract and intermediate (concrete).

Definition 2.6.6 (Abstract and Concrete ADTs)

```
ADT_A :: (state global_state, observable, global_transition) adt_t
ADT_A ≡ (|Init = Init_A, Fin = id, Step = global_automaton do_user_op_A kernel_call_A|)

ADT_C :: (kernel_state global_state, observable, global_transition) adt_t
ADT_C ≡ (|Init = Init_C, Fin = λ((tc, s), m, e). ((tc, absKState s), m, e),
          Step = global_automaton do_user_op_H kernel_call_H|)
```

In the Fin function of ADT_C, absKState is the aforementioned projection of the private concrete state to the private abstract (and therefore observable) state.

The functions kernel_call_A and kernel_call_H call the respective main kernel entry function and extract the scheduling decision of whether to switch to UserMode or IdleMode.

2.6.4 Refinement Theorem

Refinement of seL4 is proved via forward simulation with invariants. This means that the correspondence proofs require certain invariants to hold. Proving that these invariants do in fact hold constitutes a large part of the L4.verified proof effort.

Lemma 2.6.7 *The invariants hold on both the abstract and intermediate levels.*

```
ADT_A ⊨ full_invs
ADT_C ⊨ full_invs'
```

Note that full_invs comprises all invariants on the abstract level, and similarly full_invs′ on the intermediate (concrete) level.

Recall from Definition 2.2.7 that the $\models$ notation of invariance comprises establishment of the invariants by the ADT's Init function and preservation by the ADT's Step function. While the latter has been proved by L4.verified, the former is axiomatised.

Lemma 2.6.8 ADT_C *forward-simulates* ADT_A *(if the invariants hold).*

```
fw_sim_inv ADT_A ADT_C refine_rel (full_invs × full_invs')
```

Note that refine_rel is the L4.verified refinement relation.

Proving this lemma requires proving correspondence of the Init, Step and Fin functions of the ADTs. Step and Fin are proved by L4.verified. However, L4.verified axiomatised the correspondence between both ADTs' Init functions.

Using Lemma 2.6.7 and Lemma 2.6.8 in the assumptions of Theorem 2.2.9 yields the following lemma.

Lemma 2.6.9 ADT_C *forward-simulates* ADT_A.

$$\mathsf{ADT\_C} \sqsubseteq_F \mathsf{ADT\_A}$$

After applying Theorem 2.2.6, we get the final refinement theorem.

Theorem 2.6.10 (Refinement) ADT_C *refines* ADT_A.

$$\mathsf{ADT\_C} \sqsubseteq \mathsf{ADT\_A}$$

This theorem shows refinement between seL4's abstract specification and its intermediate (Haskell) specification, which is seL4's first refinement step. This step is the focus of our work. Therefore, we do not present the second refinement step here. However, note that the top-level formalisation and proof of the second refinement step work in the same way.

2.7 Weak Memory Ordering

The purpose of this section is to give an introduction to weak memory ordering for readers not familiar with this topic.

In early multiprocessor architectures, CPUs directly accessed memory via the memory bus. The bus architecture ensured that only one CPU is able to read or write data at the same time.[5] This unambiguously serialises data accesses of different CPUs, i.e. when reading, every CPU observes all writes of all CPUs in the same order. In other words, the CPUs and memory are a distributed system that ensures *sequential consistency*. Interleaving of atomic data accesses is a valid model of such a system.

With the introduction of CPU caches, sequential consistency needed to be preserved explicitly by synchronising the caches of the different CPUs. However, with CPU caches becoming larger and multiply layered, increasingly complex mechanisms (e.g. *bus snooping* or *cache coherence protocols*) were needed in order to ensure sequential consistency. Combined with rising numbers of CPUs (and therefore caches to be kept coherent) in multiprocessor systems, ensuring sequential consistency turned out to be a serious scalability bottleneck. Consequently, manufacturers stopped ensuring sequential consistency and weakened the memory ordering of their multiprocessor architectures in order to be able to optimise them for increased scalability.

2.7.1 Total Store Order (TSO)

There are a variety of weak memory models used for today's multiprocessor architectures. Each of them offers slightly different guarantees on memory ordering. In this thesis, we focus on the *total-store-order (TSO)* memory model, which is implemented, for example, by the SPARC and x86/x64 architectures.

Note that the verified uniprocessor version of seL4 runs on the ARMv6 architecture, which implements a memory-ordering model slightly different from TSO. While this results in a minor mismatch of the architecture modelled in this thesis, we decided to focus on TSO for the following reasons: (1) It is the most widely used multiprocessor model because it is implemented in every x86/x64 system. (2) The seL4::CMK proof-of-concept implementation (Section 4.5) was done for x86.

[5] Mostly, the amount of data that could be read/written atomically was a machine word. Sometimes, only reading/writing bytes was atomic.

According to Sewell et al. [SSO+10], the sole source of memory reordering in TSO are the *store buffers*, sometimes also called *write buffers*. Each CPU has a store buffer. This is a FIFO (first-in first-out) buffer that is filled by the writes the CPU issues. These writes are later drained to the *memory subsystem*, which contains the system's memory and all caches layered on top. Reads come directly from the memory subsystem. In TSO, the memory subsystem itself is sequentially consistent. The advantage of the store buffer is that the CPU is able to continue execution after a write to a cache line that has to be fetched first. If the CPU reads from a memory address it has just written to and the write is still in the store buffer, the CPU fetches the value from it directly. This is called *store-buffer forwarding*.

2.7.2 Abstracting Away Weak Memory Ordering

It is desirable that higher-level software running on weak-memory multiprocessor hardware is provided with a sequentially consistent runtime environment. This reduces program complexity and improves portability to hardware with a different weak-memory model. When done on the right level with the right abstractions, it only introduces a negligible performance overhead. In practice, it is usually achieved with a programming discipline, combined with synchronisation primitives provided by an OS kernel or hardware abstraction layer (HAL).

With weak memory abstracted away and sequential consistency established, programs can be written and reasoned about using the interleaving model, i.e. concurrent access to shared data can be modelled as interleaved sequential access. This is the basic model behind almost all verification tools and frameworks for parallel programs.

2.8 Multiprocessor Kernel Designs

In this section, we elaborate on the challenges of multiprocessor kernel programming, on common synchronisation mechanisms and multiprocessor design choices. This information is helpful in understanding related multiprocessor OS work (Section 3.1) and the design requirements that lead to the clustered multikernel (Section 4.1). However, it can be skipped by readers already familiar with this topic.

Note that throughout this thesis, we use the term *CPU* to refer to the logical unit of execution, e.g. a unicore processor or a core within a multicore processor. As such, we also use the term *multiprocessor* instead of *multicore*, unless we explicitly need to distinguish between them.

2.8.1 Challenges of Multiprocessor Programming

Designing and implementing a multiprocessor kernel is fundamentally more complex than a uniprocessor kernel. In the latter, there is only one hardware-level thread of execution. It is possible to implement in-kernel multi-threading in a uniprocessor system, which introduces software-level concurrency into the kernel. However, synchronisation mechanisms are straight-forward to implement. For example, disabling all interrupts is sufficient to start a critical section, because without interrupts, the currently running thread cannot be interrupted to switch to another thread. Often, small kernels such as microkernels are entirely non-preemptible because the longest possible system call is still shorter than the maximally tolerated interrupt latency.

All of this changes when going from one to multiple CPUs. In addition to possible software-level multi-threading, we now have hardware-level multi-threading. Each CPU

executes code independently from the others. In order to start a critical section, it is not sufficient anymore to disable interrupts. With interrupts disabled on the local CPU, another CPU could still be running the same code and access the same data structures that are supposed to be protected by the critical section. Multiple CPUs need to be coordinated explicitly, e.g. via synchronisation variables.

2.8.2 Synchronisation Mechanisms

There are a variety of synchronisation mechanisms to coordinate multiple CPUs. *Locking* is the most widely used mechanism. Normally, locks are used to protect data structures that should only be accessed by one CPU at a time. As such, the lock variable is normally co-located with the data structure it is supposed to protect. The code between the acquisition and the release of a lock is the critical section.

Another popular way of ensuring data consistency in presence of multiple CPUs is the use of *lock-free* data structures. These are data structures that are designed such that they can be read/written by multiple CPUs in a way that does not require critical sections.

Instead of accessing data structures concurrently, it is also possible to replicate or partition them between CPUs. In such a design, each CPU only accesses data structures that are assigned to it. In case of replication, CPUs have to coordinate in order to keep the replicas up to date. In case of partitioning, CPUs have to ask other CPUs to perform modifications on their behalf. This is called *message-based* synchronisation. Examples of OSes employing such synchronisation mechanisms are *K42* [K42] and *Barrelfish* [BBD+09].

2.8.3 Design Choices

When implementing a multiprocessor kernel, there are several design choices to be made. On the highest level, we have the API exported to user level. A common design pattern is to make the API mostly agnostic to how many CPUs the kernel runs on. Such an API abstracts away the presence of multiple CPUs. We say the it presents a *single-system image* to user level. Most multiprocessor kernels in the past and today use such a design. Notable exceptions are *Corey* and *Barrelfish*, which will be presented in Section 3.1.5 and Section 3.1.6, respectively. For kernels like these, CPUs are explicit resources that have to be managed from user level. For example, data structures have to be explicitly shared or replicated between CPUs; or applications have to be explicitly placed on specific CPU or distributed over certain CPUs.

On the next level below the API, it has to be chosen how the kernel data structures are accessed concurrently by multiple CPUs. For each kernel data structure, we need to decide if it is accessed by multiple CPUs at all. If yes, the question is whether this data structure should be replicated, partitioned or shared between CPUs. If shared, the choice is between lock-based or lock-free synchronisation. If we choose locks, we have to define the *lock granularity*.

Deciding on the optimal lock granularity is non-trivial. If the granularity is too coarse-grained, i.e. the critical sections too long, we jeopardise scalability. If granularity is too fine-grained, we increase locking overhead, even in the non-contended case. Fine-grained locking also has the disadvantage of increased code complexity and therefore bug density (bugs per kLOC).

Last but not least, we have to decide how to implement each lock. There are a variety of possible lock implementations. A suitable implementation has to be chosen, depending on the atomic operations provided by the multiprocessor architecture and depending on the expected degree of parallelism and contention. The most common lock implementation is

a *spin lock*, which requires an atomic TAS (test-and-set) CPU instruction. However, spin locks can cause serious scalability problems [BWKMZ12]. Scalable locks such as MCS locks [MCS91] alleviate these problems but require a more complex implementation. For example, unlike the spin lock, memory consumption of MCS locks rises with the degree of parallelism.

2.8.4 Design Requirements

When implementing a multiprocessor kernel, we have to decide on a combination of the above design choices that suits our requirements. For example, if a kernel should be optimised for scalability in a high-performance computing environment, the design choices would be very different from a kernel that is targeted at high-assurance embedded systems with a low number of CPUs. For the latter, scalability is not that important and could be traded for more security (or more flexibility).

Chapter 3

Related Work

This thesis spans three research areas: (1) multiprocessor OSes, (2) OS verification, and (3) verification of multiprocessor systems. We present related work in each of these areas.

We show that none of the multiprocessor OSes presented Section 3.1 are formally verified, that no OS verification project presented in Section 3.2 addresses multiprocessors, and that no research presented in Section 3.3 addresses higher-level verification properties such as functional correctness.

3.1 Multiprocessor OSes

Multiprocessor hardware and therefore OS kernels have been around for decades. The most widely used kernels nowadays (e.g. Windows, Linux, OS X, Unix) all support multiprocessors. They are large, monolithic kernels with sizes in the order of millions of LOC.

In this section, we focus on smaller kernels that are potentially within reach of contemporary verification techniques. In addition, we present related work on clustered multiprocessor OSes (Section 3.1.7).

We assume the reader is familiar with the necessary background on kernel architectures and multiprocessor kernel designs, which we presented in Section 2.4 and Section 2.8, respectively.

3.1.1 Mach

The original *Mach* microkernel [ABB+86] was developed at the Carnegie Mellon University as a replacement for the traditional UNIX kernel. From the start, it was designed for multiprocessor systems and to be easily portable. The API offered tasks, threads and message-based IPC via *ports*. However, unlike more recent microkernels, device drivers were part of the kernel. The microkernel approach simplified multiprocessor synchronisation. Kernel data was protected by coarse-grained locking. For example, the shared runqueue was protected by global lock.

In order to speed up IPC transfer of large messages, the data was not copied directly. Instead, the respective memory pages were remapped in a copy-on-write style. However, despite all these efforts, Mach's slow IPC speed remained the main reason for its bad overall performance. It got worse over time as CPU speed increased much faster than memory speed, because Mach had a large cache footprint.

3.1.2 Raven

The *Raven* kernel [Rit93, RN93] was built at the University of British Columbia. It was a proof-of-concept microkernel running an OS personality using user-level scheduling via scheduler activations. Raven was targeted at multiprocessor platforms with a small number of CPUs. IPC was implemented at user level via shared memory in order to avoid the cost of CPU-mode switching. The user-level threading library directly communicated with a shared memory section within the kernel. In-kernel synchronisation appears to have been fine-grained as the authors mention multiple locks. Raven was not designed for scalability and did not scale beyond a handful of CPUs.

3.1.3 QNX

QNX Neutrino RTOS [QNX] is a commercially deployed, microkernel-based real-time OS. It is tailored to embedded systems and supports multiple architectures.

The user can choose between three multiprocessing models: (1) symmetric multiprocessing (SMP), (2) bound multiprocessing (BMP) and (3) asymmetric multiprocessing (AMP). SMP offers full flexibility in terms of load balancing between CPUs. In order to increase affinity of processes to CPUs, BMP enables the user to bind processes to certain CPUs. Legacy uniprocessor code is supported by the AMP model, where CPUs can be dedicated to run legacy code that is not directly managed by the microkernel. BMP and AMP are potentially more scalable than SMP because they explicitly assign resources to CPUs which therefore do not require synchronisation.

Within the microkernel, synchronisation is done with a big lock, i.e. only one CPU is allowed to run in the kernel at the same time.

3.1.4 L4-based Kernels

Long before the first multiprocessor version of L4 existed, Liedtke briefly discussed the implications of multiprocessor support [Lie93] and argued that locking will have a significant impact on all performance-critical operations.

L4/Alpha

The first multiprocessor implementation of L4 was done for the Alpha architecture [Pot99, PWH02]. The focus of this research was on scalability, i.e. data locality and performance of CPU-local operations. Cross-CPU operations used slower message-based synchronisation.

Fiasco-SMP

Around the same time, Hohmuth et al. [HH01, Hoh02b] presented *Fiasco-SMP*, a multiprocessor version of their real-time L4 kernel *Fiasco*.

Fiasco is fully preemptible and synchronisation between CPUs is wait-free. To this end, Fiasco implements a *multiprocessor priority-inheritance protocol (MPIP)*, which is one of the main contribution of the authors' work. The MPIP, also called *helping mechanism*, works as follows: If a higher-priority thread needs access to a resource blocked by a lower-priority thread running on another CPU, the lower-priority thread is temporarily migrated to the higher-priority thread's CPU and is given the remainder of that thread's timeslice.

The helping mechanism extends to user level via the kernel API. This allows threads to explicitly donate the remainder of their CPU timeslice to a server thread which they are requesting a service from, no matter which CPU the server thread is currently running on.

The helping mechanism improves real-time behaviour. However, in case of multiprocessors, it leads to excessive thread migration between CPUs and thereby has a negative overall performance and scalability impact [Uhl05].

L4Ka::Pistachio

Uhlig [Uhl05] investigated scalability of microkernel-based systems. He argued that microkernels should preserve the application's level of parallelism and data locality.

To this end, Uhlig contributed a multiprocessor implementation of *L4Ka::Pistachio* that dynamically adapted the synchronisation mechanism (lock-based or message-based) and the locking granularity according to the current application's level of parallelism. Furthermore, he applied the concepts of RCU (read-copy update) synchronisation [MS98] to implement a scalable TLB-invalidation protocol and to manage the dynamic changes of locking granularity.

L4.sec

Frenzel [Fre06] investigated and implemented multiprocessor support for *L4.sec*, which was a capability-based L4 version developed in parallel to seL4. The focus of the thesis was on locking schemes for TCBs (thread control blocks) and communication endpoints.

As the evaluation was done on a two-CPU machine and with micro benchmarks only, no conclusions about scalability can be drawn.

OKL4

The *OKL4 Microvisor* [Ope] is a commercial microkernel/hypervisor. It has been successfully deployed in over 1.5 billion multicore mobile phones to date [OKL12]. For synchronisation in a multiprocessor environment, OKL4 uses a single big lock.

Lyons [Lyo11] performed preliminary multiprocessor benchmarks of OKL4 on a NaviEngine platform (four ARM11 MPCore CPUs). She ran one virtual machine (VM) per CPU with an instance of uniprocessor Linux running in each VM. The workloads to be run within Linux were chosen from lmbench and SPEC2000 [SPE00].

For the benchmarks, OKL4 was extensively instrumented to measure how often each system call was invoked and how long its execution took. Furthermore, the big lock was instrumented to measure contention and waiting time.

An interesting finding was that the most intensively used system calls do not need to acquire the lock because they only work on CPU-local data. Regarding the remaining system calls, Lyons concluded that on the NaviEngine platform, "cache contention has a much higher impact on scalability than locking scheme". In other words, fine-grained locks would not remove the scalability problem.

3.1.5 Corey

Corey [BWCC+08] is a small multiprocessor OS based on the Exokernel [EKO95] approach. It is designed to increase scalability by reducing sharing. Specifically, the kernel allows applications to specify sharing requirements for kernel data.

The basic idea is to assign OS services and applications to specific CPUs in order to reduce sharing between CPUs. As such, the kernel's default behaviour is to arrange its data structures such that they are always accessed by the same CPU. However, applications are allowed to specify that certain kernel data structures should be shared between certain

CPUs. Applications are also free to share user-level memory between CPUs. For kernel data structures that are shared, Corey uses fine-grained MCS locks [MCS91].

Benchmarks show significant performance improvements over Linux for a higher number of CPUs (up to 16 CPUs were tested). However, as the authors admit, Corey is not a fully fledged OS and lacks features that are present in Linux that make it operate slower. Nevertheless, the results suggest that the monolithic design of Linux is an inherent scalability bottleneck that can only be overcome with a new OS design (such as Corey).

In later work [BWCM+10] however, the authors come to the opposite conclusion. They substantiate it by iteratively repeating the following process: (1) running their *mosbench* [mos] Linux scalability benchmark, (2) identifying the main bottleneck, and (3) fixing the bottleneck. The bottlenecks were fixed by using standard parallel-programming techniques. Often, the bottlenecks were global counter variables used for reference counting. In these cases, the authors applied a new technique they call *sloppy counters*. A sloppy counter removes the need for accessing a global variable on each increment/decrement. Instead, each CPU holds a few "spare references" in a variable of its own. Unless these references are used up, increasing the reference count is a CPU-local operation. Only when they are used up, the CPU has to access (increment) the global counter variable in order to acquire new references. Similarly, decreasing the reference count is a CPU-local operation. Only if the CPU's number of spare references grows beyond a threshold, they are returned to the global counter variable by decrementing it accordingly.

By removing multiple bottlenecks, the authors increased the scalability of Linux considerably. Furthermore, they found nothing that would prevent further efforts in removing more bottlenecks and therefore increasing scalability even further.

3.1.6 Barrelfish

Baumann et al. [BBD+09, BPS+09, SPB+08] present the *multikernel* design, a new way of designing multiprocessor OSes with better support for heterogeneous multiprocessor hardware with a high number of CPUs. Along with the multikernel design, the authors present *Barrelfish*, a proof-of-concept multikernel OS.

The main idea behind the multikernel approach is to treat the underlying hardware as a distributed system. Instead of sharing kernel and application data between CPUs, data is partitioned or replicated. Synchronisation takes place via message-passing between CPUs.

In Barrelfish, each CPU runs its own instance of the Barrelfish kernel (called *CPU driver*), which is a microkernel whose API is inspired by seL4. As such, no kernel data structures are shared between CPUs. On top of the kernel, a *monitor* coordinates the message-based synchronisation of replicated kernel data structures, such as page tables.

Similar to Corey, the multikernel approach leads to system designs in which OS services and applications are assigned to specific CPUs. While this results in better memory locality, there is a danger of specific OS services becoming a (temporary) bottleneck under certain workloads.

The clustered-multikernel design contributed by this thesis (Chapter 4) is an extension of the multikernel design of Baumann et al.

3.1.7 Clustered OSes

Clustered OS kernels emerged in the early 90s.

Hurricane

Hurricane [UKGS93] used clustering to improve data locality on large-scale NUMA multiprocessors.[1]

In Hurricane, the kernel provided address-space management, processes, scheduling, IPC and device drivers. The API level is therefore comparable to a monolithic kernel. However, the file system was largely implemented at user level.

Behind its API, the Hurricane kernel was clustered. A cluster spanned a configurable set of CPUs. Kernel data was not shared between clusters. Instead, it was partitioned or replicated with message-based synchronisation. Within a cluster, fine-grained locking was used. As such, clustering provided a configurable tradeoff between contention and communication overhead.

The evaluation was targeted at measuring system performance depending on the cluster configuration. The benchmarks were run on a Hector 4x4 CPU NUMA machine. All possible clustering configurations (cluster sizes of 1, 2, 4, 8 and 16 CPUs) were tested.

The benchmarks included operations on shared resources, e.g. multiple processes communicating with the same specific process, or all processes faulting on the same set of pages. As a result, performance increased with increasing cluster size, peaking at a cluster size of 8. After that, performance decreased again. They concluded that for operations on shared resources in general, there is an optimal intermediate cluster size. The exact number depends on the operation.

Running "real-world" benchmarks such as matrix multiplication and 2D-FFT (fast Fourier transform) resulted in the performance being best with a cluster size of 4.

Despite many benefits of Hurricane, the authors later found [GKAS99] that Hurricane's "rigid clustering" resulted in performance problems for certain kinds of applications, mainly due to communication overhead between clusters and poor data locality within clusters.

In Section 4.4.1, we will relate this work to our clustered-multikernel design in more detail. We will also elaborate on why we believe that Hurricane's problems do not necessarily arise in our design.

Hive

Hive [CRD+95] was a distributed OS with independent kernels called *cells*. Hence, it had a structure very similar to Hurricane. Hive implemented coordination between cells in order to provide a single-system image to the user.

The research focus was on fault isolation between cells. More precisely, they added a "firewall" feature to the memory controller of the Stanford FLASH multiprocessor system they used to evaluate Hive. The firewall was a per-page write-permission bit vector. It was used to protect cells from potential wild writes of other cells. The challenge was to do this in presence of resource sharing between cells. In order to solve the problem, resources were allocated in a way that a faulting cell could not impact other cells.

Since the required FLASH multiprocessor hardware was not available, they had to simulate it in order to evaluate Hive. Simulated benchmarks were run on three configurations: one cell of four CPUs, two cells of two CPUs, and four cells of one CPU. As a baseline for comparison, the IRIX 5.2 SMP OS was used.

The results were unsurprising: For workloads with a low system-call rate, the overhead over IRIX was negligible, regardless of the cell configuration. In case Hive was configured

[1] In NUMA (non-uniform memory access) systems, physical memory is partitioned. Each partition is assigned to a cluster of CPUs. Memory accesses within a cluster are much faster than across clusters.

with only one cell spanning all CPUs, the overhead was negligible for any tested workload. In all other cases, the overhead was around 10%.

Similar to Hurricane, Hive suffered from high complexity.

Disco

In later work, Hive's authors presented *Disco* [BDR97], which aimed at reducing implementation cost and complexity by reviving the idea of virtualisation. Note that in the years before Disco, virtualisation was not in the focus of the OS research community anymore: Interest in virtualisation declined in the 80s after its initial rise in the 70s.

The idea behind the virtualisation approach was to enable commodity OSes that were not designed for large-scale NUMA multiprocessor systems to run on such systems, without the additional complexity and effort of having to modify them accordingly. In Disco, the "unit of scalability" was the VM, akin to a cell in Hive and a cluster in Hurricane. Similar to a cell in Hive, the VM was also the "unit of fault containment".

Disco was a monolithic multiprocessor virtual-machine monitor (VMM) for the Stanford FLASH multiprocessor architecture. For synchronisation between CPUs, it mostly used wait-free data structures besides a few locks. A page-migration mechanism allowed transparent, NUMA-aware migration of host physical memory between virtual machines (VMs), depending on memory-access statistics. Sharing copy-on-write memory between VMs was supported as well. Communication between VMs was possible via a shared virtual network. All hardware devices where virtualised by the VMM and could therefore be shared between VMs.

For the evaluation, the IRIX 5.3 OS was run on top of Disco. Virtualisation overhead was reduced by small changes to the hardware abstraction layer (HAL) of IRIX, a technique that would be called *paravirtualisation* today. Like for Hive, the authors had to simulate the Stanford FLASH multiprocessor system because it was not available in hardware. In contrast to the four CPUs for Hive, the machine simulated for Disco had eight CPUs.

For benchmarking, they ran multiple workloads in the IRIX VMs, e.g. raytrace and pmake (a parallel version of make). First, they measured the raw single-CPU virtualisation overhead, which was 3% for raytrace and 16% for pmake. The low overhead for raytrace is because it is computationally bound, i.e. the VMM is invoked rarely. In contrast, pmake performs a lot of I/O and address-space creations/destructions, which explains the higher virtualisation overhead.

As scalability benchmark for raytrace, they compared running it on a single, native eight-CPU IRIX instance vs. running it distributed over eight IRIX VMs on Disco with each VM running on one CPU. On the latter, it ran 22% faster, despite the virtualisation overhead. For pmake, they also tried intermediate VM configurations, i.e. the available CPUs were divided up between one, two, four and eight VMs. The first configuration reveals the single-CPU virtualisation overhead. With two VMs, pmake ran 8% faster in Disco than on native IRIX, despite the virtualisation overhead. In the eight-VM configuration, the speedup was 40%.

While such a speedup looks impressive at first sight, it has to be seen in relation to its base line: IRIX did perform neither NUMA-aware memory allocation nor dynamic page migration. Furthermore, IRIX had well-known scalability bottlenecks [BDR97].

Tornado/K42

In the light of the already mentioned problems of Hurricane, its authors realised that more fine-grained tuning of data locality was needed. This lead to *Tornado* [GKAS99] and its

successor project *K42* [K42].

The approach was to construct the kernel in an object-oriented manner. The novel idea of *clustered objects* allowed kernel data locality to be fine-tuned by partitioning objects into *representative objects (reps)*. A rep spans a configurable number of CPUs. When services are requested from an object, each rep can service a request, independently from the other reps. Reps communicate with each other (via shared memory) in order to present a single object instance to the user. There were no global locks in Tornado/K42. In case of a rep spanning multiple CPUs, rep-local locks were applied if necessary. Another key feature of Tornado/K42 was that object implementations could be changed at runtime, depending on scalability needs.

Tornado/K42 was implemented in C++. Initially, it ran on a simulated 4x4 CPU MIPS NUMA machine. Later, it was ported to the PowerPC architecture.

In the evaluation, Tornado/K42's performance/scalability was compared to a number of general-purpose OSes of that time and performed better than those. However, it was not compared to a clustered OS such as Hurricane.

3.1.8 Analysis

The OSes presented in the above sections employ a variety of multiprocessor designs: A standard locking approach was taken by Mach, Raven, L4/Alpha, L4Ka::Pistachio, L4.sec, and within clusters of Hurricane. L4Ka::Pistachio additionally uses message-based synchronisation. A simple big-lock approach was taken by QNX and OKL4. Wait-free synchronisation is used in Fiasco-SMP and within clusters of Disco. Corey uses partitioning of kernel data while enabling user level to specify explicit sharing. Barrelfish is based on replication and message-based synchronisation of kernel data structures. Hurricane, Hive and Disco use clustering, i.e. they replicate kernel data in clusters which are aligned with NUMA clusters in order to increase data locality on NUMA machines. Clustering in Tornado/K42 is more fine-grained, i.e. on the level of kernel objects instead of NUMA nodes.

Most OSes presented here employ complex multiprocessor designs. We believe that the combination of this complexity with the kernels' functional complexity makes formal verification intractable. Notable exceptions are the big-lock design used by QNX and OKL4 and the in-kernel-part (CPU driver) of the multikernel design implemented in Barrelfish. These two designs will be discussed in detail in Section 4.2 and Section 4.3, respectively.

3.2 OS Verification

In this section, we present related work on OS verification. We focus on research that aims at verification down to the implementation level and that uses machine-checked models and proofs (as opposed to pen-and-paper only).

None of the research presented in this section addresses multiprocessors.

3.2.1 Early Work

The history of OS kernel verification starts in the 70s and 80s. The *Provable Secure Operating System (PSOS)* [FN79,NBF+80] was a hardware/software co-design project with the goal of "a useful general-purpose operating system with demonstrable security properties" [NF03]. Even though no actual code proofs had been undertaken, the project pioneered important

concepts that facilitate OS verification, such as encapsulation and information hiding. They also aimed at applying formal methods throughout the entire implementation process.

UCLA Secure Unix [WKP80] was a kernel that provided services similar to modern microkernels. It was implemented in a simplified version of Pascal. The verification project aimed at showing data refinement (see Section 2.2), although at the time it was not called by that name. Above the implementation level, there were three specification levels. The goal was to prove refinement between the top-level specification and the implementation. The authors report that about 90% of the specification and about 20% of the code proofs had been completed.

KIT [Bev89] was a small kernel that comprised about 300 LOC of assembly. It provided task isolation, I/O and single-word message passing. KIT was formally verified down to the assembly level with a data-refinement approach very similar to the one used for UCLA Secure Linux. Despite the small size of KIT, this was a significant achievement. KIT conclusively demonstrated that the "level of detail required in OS implementation verification is not an intrinsic problem for formal verification." [Kle09].

In the decade after KIT, no larger-scale, serious attempts at kernel verification were undertaken. In the years after 2000, the topic attracted new interest. We present the resulting research projects hereafter.

3.2.2 VFiasco

The *VFiasco* project [HT05, HT03] and its successor *Robin* [TWV08] aimed to verify the *Fiasco* microkernel [Hoh02a] and its successor *NOVA* [SK10] directly on the level of C++ using the *PVS* theorem prover [ORR+96]. They developed a precise model of a large subset of C++ deemed necessary but did not verify substantial parts of these microkernels before the projects ended.

3.2.3 Coyotos

Shapiro et al. [SDN+04] aimed at formally verifying the *Coyotos* microkernel, which is the successor of the capability-based *EROS* microkernel [SSF99], which in turn, is inspired by the *KeyKOS* kernel [Har85] and *CAP* computer [NW77].

The main focus of their research was the new programming language *BitC*, which is tailored to low-level system programming and verification at the same time. For example, it allows breaking type-safety in a controlled way. It is unclear what progress has been made on the verification of Coyotos itself.

3.2.4 Verisoft

The *Verisoft* project [AHL+08, Vera] achieved considerable success in pervasively verifying a whole computer system. Pervasive verification means that there is a formally connected proof that covers all layers of a computer system, from the hardware up to the application.

In Verisoft, the lowest layer is the *VAMP* hardware platform [BJK+06], which implements its own ISA (instruction set architecture). The behaviour of the ISA has been verified down to the gate level.

The software stack is implemented in the proprietary language *C0* for which they provide a non-optimising compiler with verified back-end [LP08]. The bottom layer of the software stack is called *CVM* (communicating virtual machines) [dRT08], which is a hardware abstraction layer (HAL) on which the *VAMOS* microkernel [DDW09, ST08]

runs. The OS personality implemented on top of VAMOS is called *SOS* (simple operating system).

All layers of the Verisoft project have been implemented and formally specified. For most of them, pen-and-paper proofs exist. Roughly 75% of the proofs are machine-checked in the interactive theorem prover Isabelle/HOL [NPW02]. For the VAMOS microkernel, a large part of the machine-checked proofs has been completed [Kle09]. The Verisoft project did not investigate higher-level properties such as isolation.

3.2.5 OLOS

OLOS [Sch11, DSS09] is a simple time-triggered real-time OS kernel, implemented on top of Verisoft's CVM HAL and therefore running on the VAMP architecture. The implementation in C0 comprises roughly 300 LOC.

Its functional correctness is formally verified in the context of Verisoft's pervasive verification. This means that there is a refinement proof in Isabelle/HOL that connects an abstract specification of OLOS to the implementation and the lower levels of Verisoft's system stack (CVM, VAMP).

3.2.6 seL4/L4.verified

We reported on the formal verification of the seL4 microkernel [EDE08]—the L4.verified project [KEH+09]—in Section 2.6.

3.2.7 Verve

Verve [YH10] is a research OS developed by Microsoft. Similar to CVM/VAMOS, it consists of a HAL (called *Nucleus*) and the actual kernel, which is implemented on top of the Nucleus. Applications run on top of the kernel. As with CVM/VAMOS, dividing up the kernel functionality into two layers aids verification because high-level typed code can be reasoned about separately from low-level untyped code.

The Nucleus provides memory allocation, stack switching, device access and IRQ management. It is written in annotated assembly language which is verified against a specification of correctness and safety by *Boogie/Z3* [BCD+06]. On top of the Nucleus, the kernel implements preemptive threads, scheduling, synchronisation and IRQ handlers. The kernel and all applications running on top of it are written in managed C# and compiled to TAL (typed assembly language) by *Bartok* [CHP+08]. A TAL checker then automatically verifies type safety of the compiled code directly on the assembly level.

Together, this proves type safety and memory safety of the entire system because (1) the Nucleus is proved to be type and memory safe regardless of the kernel's requests, (2) the kernel is proved to be type safe, and (3) the kernel can only allocate memory via the Nucleus. However, type and memory safety do not imply functional correctness. A scheduling bug, for example, can still go undetected and make the system misbehave.

3.2.8 Analysis

OS kernel verification is a very complex and labour-intensive undertaking, even using the latest verification techniques, and even when restricted to uniprocessor support. Successfully verifying an entire kernel down to the implementation level only seems possible with a coordinated large-scale verification project (seL4), with a very small kernel

(OLOS), or when aiming for lower-level properties such as type and memory safety (Verve).

In this thesis, we aim for formal implementation-level verification of a general-purpose multiprocessor kernel.

3.3 Verification of Multiprocessor Systems

Properties about multiprocessor computer systems can be verified on a variety of levels. At the lowest level—the hardware—multiprocessor-specific verification efforts mostly concentrate on correctness of cache coherence protocols [PD97]. Manufacturers have been routinely employing formal verification techniques to verify the correctness of their CPUs, including the cache coherence protocols.

The levels above the hardware have received considerably less attention, especially from industry. However, correct specification and use of the interface between the multiprocessor hardware and the software running on top of it is crucial. This aspect had been neglected during the advent of weak memory ordering, as shown by the *x86-TSO* work presented below in Section 3.3.3. Further work attempting to close this gap includes the *TSO reduction theorem* (Section 3.3.4) and our formal TSO model that we will contribute in Section 5.2.2.

Focus The verification property this thesis focuses on is functional correctness of concurrent program code. As such, this section concentrates on related work that aims at parallel-program verification using theorem proving. In contrast, we consider model checking of concurrency primitives or protocols only marginally related. This includes, for example, proving deadlock freedom or fairness of locking algorithms.

3.3.1 VCC

Microsoft's *VCC* verification framework [CDH+09] allows reasoning about C code, including concurrent code and low-level kernel code. The basic concept is that the desired properties to be proved are formulated as C code annotations, e.g. asserts or function contracts (pre- and postconditions). Predicates used in these annotations must be formulated in first-order logic and may contain quantifiers. For each function, VCC tries to automatically prove the correctness of the annotations and—for each function call—whether the annotated function contracts are adhered to.

The VCC approach is an important contribution to concurrent code verification. It is the first to enable reasoning directly about concurrent C code while providing high automation. It manages to keep verification tractable in most cases, even for highly complex code.

Workflow

VCC uses Boogie/Z3 [BCD+06] to convert the annotations into proof obligations and tries to discharge them automatically. There are three possible outcomes: (1) The proof succeeds, i.e. we have proved that the annotations are correct. (2) The proof fails. In this case, VCC tries to translate the failed proof obligation back to a meaningful counterexample in C in order to help fix the problem. (3) There is a timeout, i.e. VCC neither finds a proof nor counterexample within a configurable time span. For this case, VCC provides the possibility to inspect the proof state (e.g. quantifier instantiations) in order to find out

why a proof could not be found. It is also possible to guide VCC in finding a proof by, for example, suggesting quantifier instantiations.

In addition, the *HOL-Boogie* framework [BLW08, BMSW10] allows problematic proof obligations to be discharged by a manual proof in Isabelle/HOL.

Ghosts

A crucial concept of VCC is the use of *ghost* state/code for abstraction. Ghost state/code is routinely used in program verification. It is usually written in the same programming language as and mixed with the operational program state/code. However, unlike the latter, it does not influence program execution and is ignored by the compiler. As such, ghost code is only allowed to read/write ghost state and read operational state. It is not allowed to write operational state.

VCC supports ghost state/code written in a superset of the C language, which can be used for abstraction. For example, there is a ghost type that defines a set. Therefore, an array in the operational state can be abstracted as a set by maintaining a *shadow copy* of that type in ghost state. The copy is maintained by ghost code updating the set whenever the array is updated. The crucial part of this concept is that VCC allows ghost state to be reasoned about in annotations. VCC ensures that no information flows from ghost state to operational state.

In VCC, ghost state includes: ghost variables, ghost fields (in operational structs) and ghost parameters (of operational functions). The user is also allowed to define new ghost types.

Type Safety

Besides assert and function-contract annotations, VCC also supports *type invariants*. For each C type, the user can define invariants whose preservation VCC tries to prove automatically. Furthermore, VCC allows *weak typing*. This means that type information is kept in a ghost *typestate*. Type safety can be proved automatically, and wherever type safety is broken, dynamic type information can be annotated.

Concurrency

Usually, large parts of concurrent programs are actually sequential, i.e. the code reads/writes a subset of the state that is considered thread-local at the time of access. Thread locality can either be static or dynamic. For example, static thread locality is given (1) for local variables (which reside on the stack) and (2) for variables defined in TLS (thread-local storage). Examples of dynamic thread locality are (1) shared data protected by a lock or (2) temporarily allocated data for modification of lock-free data structures.

VCC uses this fact to reduce verification complexity by introducing an *ownership* discipline. It provides built-in ghost operations for acquiring and releasing temporary ownership of shared state. In the simple example of a mutex lock, these operations coincide with the operational acquire/release of the lock. VCC proves that shared state is only owned by a single thread at any point in time. In return, it can reason about the accessing code in a completely sequential way. Furthermore, type invariants are allowed to be temporarily violated while ownership is held. They have to be established again before releasing the ownership.

In C programs, variables that are accessed concurrently by multiple threads have to be declared as `volatile`. This prevents the compiler from caching the variable in CPU

registers. Examples of such variables are locks or pointers in lock-free data structures. In this light, VCC assumes that the programmer declares a variable as `volatile` if and only if it is (potentially) accessed concurrently. Volatile variables are treated specially by VCC: (1) Invariants need to be preserved by each update of such a variable. (2) Invariants talking about volatile variables have to be formulated as *claims*, which are ghost objects. As such, they can be created/destroyed and references to them can be passed around. For example, passing such a reference to a function as ghost parameter serves as a stronger kind of precondition: It is not restricted to thread-local variables and the claim invariant is guaranteed to hold until the claim is destroyed.

VCC also supports *two-state invariants*. These invariants talk about the values of variables before and after an atomic update. To this end, VCC defines the ghost function `old()`, which refers to the value of a variable before an atomic update. As an example, the invariant `x > old(x)` requires that whenever the variable `x` is written to, the new value is larger than the old value. Two-state invariants can be used for Rely-Guarantee-style reasoning.

3.3.2 Verisoft XT

Hyper-V

The *Verisoft XT* project [Verb] used VCC for formally verifying *Hyper-V*, which is a commercially widely deployed multiprocessor hypervisor by Microsoft. Hyper-V consists of 100 kLOC of concurrent C code and 5 kLOC of assembly code. It runs on x64[2] multiprocessor machines with Intel or AMD virtualisation extensions.

The authors report that about 20% of Hyper-V's C code has been verified [CDH+09]. After having been fully annotated, most functions required between 0.5 and 500 seconds to automatically verify. However, in many cases, the verification performance was still unacceptable, e.g. verifying some functions took several hours or days.

The properties that have been proved are mainly function contracts and type invariants. Unfortunately, these results are not sufficient to conclude an overall functional-correctness theorem from. Moreover, the VCC annotation language is very close to C, which leads to low-level specifications. Their level of detail makes it hard to prove higher-level properties such as isolation or integrity.

PikeOS

Verisoft XT also used VCC to tackle the formal verification of *PikeOS*, an L4-based commercial separation kernel by SYSGO. PikeOS comprises about 6000 LOC of C and supports multiple architectures.

Baumann et al. [BBBB09, BBBB10] mainly introduce the verification framework and methodology. The actual code proof only covers one simple system call. In a later paper [BBBT11], the authors report having proved memory separation for a PikeOS memory manager.

Formal verification of PikeOS is restricted to a uniprocessor setup. However, the authors mention that the verification methodology should be extensible to cover a multiprocessor setup.

[2] Throughout this thesis, we use the term *x86* for the *IA-32* and *AMD32* architectures, and the term *x64* for the *Intel 64* and *AMD64* architectures.

Baby Hypervisor

In another subproject of Verisoft XT, VCC was used to formally verify a simple hypervisor called *baby hypervisor* on the C and assembly level [AHPP10, PSS12]. This hypervisor runs on a hypothetical simple 32-bit RISC architecture called *baby VAMP*, which is formalised in VCC using ghost state/code. The proof states that the hypervisor simulates the execution of the guest correctly, i.e. the guest cannot observe any difference between executing in the native vs. the virtualised environment.

Baby VAMP is a uniprocessor architecture and there is no thread-level parallelism in the baby hypervisor. Consequently, the proof does not have to deal with concurrency. However, the authors mention that in future work, a multiprocessor version of the baby hypervisor could theoretically be verified with VCC using the same approach.

This project played an important role in driving the development of VCC.

3.3.3 x86-TSO

Sewell et al. [SSO+10, OSS09] identify that manufacturers of multiprocessor systems often specify their weak memory model insufficiently. For example, architecture programming manuals specify in ambiguous informal prose what programmers can rely on. Focussing on x86, Sewell et al. analyse the AMD and Intel Developer Manuals [AMDb, Inta] and find that they contain serious ambiguities, and in some cases, are even unsound with regards to the actual hardware. Remember that x86 implements TSO memory ordering, which we explained in Section 2.7.1.

In order to alleviate this situation, Sewell et al. present a formal *x86-TSO* model which suffers from none of the above mentioned problems. It is formalised in the interactive theorem prover HOL4 [HOL]. Ambiguities are resolved by informed guesses according to the "folklore in the area" and by what behaviour can be observed in practice. For this, the authors run extensive empirical tests on actual hardware.

Their results "strongly suggest" that apart from the buffering in the CPUs' store buffers, all CPUs share the same view of memory. In other words, the memory subsystem (the memory with all caches layered on top) is sequentially consistent and the sole source of memory reordering are the store buffers. Furthermore, they conclude that executing a memory fence flushes the CPU's store buffer, even though the Intel Developer Manual is vague about this.

These results are significant. The authors are the first to identify that x86 is a TSO architecture, even though not specified as that by the manufacturers. As a result of the findings in this work and the authors' previous work [SSN+09], the manufacturers corrected the identified mistakes and ambiguities in their manuals.

Formal Model

The x86-TSO model of Sewell et al. consists of two equivalent definitions: (1) an *abstract machine* with explicit store buffers, and (2) an *axiomatic model* which defines valid executions in terms of memory orders.

The former is defined in *operational semantics*, which means that the behaviour of the system is defined by explicitly modelling the involved hardware parts (CPUs, store buffers, memory subsystem) and how they interact with each other. In contrast, the latter uses *axiomatic semantics*, which describes the behaviour of the system as a set of rules on its input/output. For example, informal axiomatic semantics is used to describe memory

ordering in the AMD and Intel Developer Manuals [AMDb,Inta]: "stores are not reordered with other stores", "writes cannot pass `MFENCE` instructions", etc.

The abstract machine explicitly models the CPUs, their registers and store buffers, the memory subsystem and the x86 locking mechanism. The latter enables certain read-modify-write instructions to be made atomic by adding a `LOCK` prefix. For example, `LOCK INC` executes an atomic increment directly in memory.

The axiomatic model is based on the SPARCv8 memory-model specification [SPA92] and adapted to x86. It is a formalisation of the memory reordering rules found in the developer manuals with ambiguities resolved as mentioned before.

One of the main contributions of this work is a proof (in HOL4) that the abstract machine and the axiomatic model are equivalent.

In a later paper, Ridge [Rid10] contributes a Rely-Guarantee-style proof system (in HOL4), based on the x86-TSO model.

Limitations

The x86-TSO model only applies to x86 memory configured as *write-back* (which is the default), because other x86 memory types do not implement TSO. For example, *uncacheable* memory exhibits sequential consistency while *write-combined* memory provides less ordering guarantees than TSO. Furthermore, the model does not cover CPU exceptions, misaligned or mixed-size memory accesses, self-modifying code and page-table changes.

3.3.4 TSO Reduction Theorem

Cohen et al. [CS10,CS09] present a reduction theorem for TSO architectures, formalised and proved in Isabelle/HOL [NPW02]. The theorem states that a parallel program running on a TSO machine observes a sequentially consistent machine if the program adheres to a certain programming discipline.

As already mentioned in Section 3.3.1, there are common patterns of concurrency control that parallel programs normally adhere to. For example, data is protected by a lock and can be read/written without further concurrency control as long as the lock is held; or data is shared and modified concurrently with the help of atomic instructions (e.g. lock-free data structures). These concurrency-control disciplines rely on the sequential consistency of the underlying system (multi-threading provided by the OS or multiple CPUs provided by the hardware). Therefore, when programming for TSO multiprocessor machines, programmers usually employ further programming disciplines to regain sequential consistency.

Programming Disciplines

An example of a naive discipline is to place a memory fence after every write to a shared variable. Unfortunately, such disciplines do not perform well enough to build concurrent high-performance software. Therefore, Cohen et al. present a programming discipline that aims at maximising performance by minimising the number of volatile memory accesses and the number of store-buffer flushes.

The discipline is based on an ownership principle. Like in VCC, ownership can be static or dynamic. It is tracked in ghost state and transferred though ghost operations. As such, it incurs no compile or runtime overhead. Memory reads/writes need only be volatile if the memory address is unowned/shared. In case the address is owned/unshared, normal non-volatile reads/writes can be performed. It is also possible to declare memory

addresses as owned/shared, which covers the special case of data structures with a single writer (owner) and multiple readers. In this case, the owner is allowed to perform non-volatile reads/writes while the readers are required to use volatile reads. This minimises the number of volatile reads/writes.

Store-buffer flushing works as follows: For every thread, a *dirty* flag is maintained in ghost state. It is set after a write, which models that the store buffer becomes non-empty (dirty). The flag is cleared again after a store-buffer flush. In order to regain sequential consistency, it is sufficient that the store buffer is flushed only before a volatile read and only if the dirty flag is set.

Formalisation

In order to be able to formulate the TSO reduction theorem, the authors provide a formal model of a *store-buffer machine* and of a *virtual machine*.

The former is similar to the x86-TSO abstract machine (Section 3.3.3). Memory content is modelled by each memory address representing a variable which is read/written independently from other variables. The model includes the following operations: (1) reading from memory, (2) reading from the store buffer (store-buffer forwarding), (3) writing to the store buffer, and (4) store-buffer content being written to memory. Read and write operations are tagged as volatile or non-volatile. Program execution is modelled with a parallel variant of IMP [Nip98], which is a simple imperative programming language with WHILE loops.

The *virtual machine* is the same as the store-buffer machine, except that it is sequentially consistent, i.e. it has no store buffers. In addition, it maintains ghost state/code that checks if the programming discipline is adhered to.

The TSO reduction theorem states that if the programming discipline is adhered to—which needs to be proved over the virtual machine's ghost state—then every computation of the store-buffer machine can be simulated by a computation of the sequentially consistent virtual machine.

3.3.5 Analysis

Even though not completed, the Hyper-V project showed that formal implementation-level verification of a complex, large multiprocessor hypervisor is within reach of contemporary verification techniques. However, this is only true when targeting lower-level properties such as function contracts or type invariants. The fact that proving higher-level properties such as functional correctness has not been or unsuccessfully been attempted suggests that proving such properties about large, complex hypervisors or kernels is still intractable.

The PikeOS and Baby Hypervisor projects only targeted uniprocessors, although the authors mention that the chosen verification approach can theoretically be used to prove multiprocessor versions of these systems as well. However, it is unclear whether the approach remains tractable in the multiprocessor case, especially for a general-purpose kernel such as PikeOS.

In contrast to this thesis, none of these verification projects address weak memory ordering. The presented related work on weak memory ordering (x86-TSO and the TSO Reduction Theorem) does not directly aim at kernel verification. Instead, x86-TSO contributes an *abstract machine* and an *axiomatic model* of TSO as implemented by x86, and a proof that they are equivalent. The TSO Reduction Theorem work contributes a *store-buffer machine* (which is very similar to the x86-TSO abstract machine) and a sequentially

consistent *virtual machine* without store buffers. The theorem states that if a given parallel-programming discipline is adhered to, every computation of the store-buffer machine can be simulated by a computation of the virtual machine.

The TSO model contributed in this thesis (Section 5.2.2) is inspired by the x86-TSO abstract machine.

3.4 Summary

We presented related work in three research areas: (1) multiprocessor OSes, (2) OS verification, and (3) verification of multiprocessor systems.

Multiprocessor OSes Most OSes presented here employ complex multiprocessor designs. We believe that the combination of this complexity with the kernels' functional complexity makes formal verification intractable. Notable exceptions are the big-lock design used by QNX and OKL4 and the in-kernel-part (CPU driver) of the multikernel design implemented in Barrelfish. These two designs will be discussed in detail in Section 4.2 and Section 4.3, respectively.

OS Verification OS kernel verification is a very complex and labour-intensive undertaking, even using the latest verification techniques, and even when restricted to uniprocessor support. Successfully verifying an entire kernel down to the implementation level only seems possible with a coordinated large-scale verification project (seL4), with a very small kernel (OLOS), or when aiming for lower-level properties such as type and memory safety (Verve).

Verification of Multiprocessor Systems Even though not completed, the Hyper-V project showed that formal implementation-level verification of a complex, large multiprocessor hypervisor is within reach of contemporary verification techniques, albeit only for lower-level verification properties.

Conclusion We showed that none of the multiprocessor OSes presented Section 3.1 are formally verified, that no OS verification project presented in Section 3.2 addresses multiprocessors, and that no research presented in Section 3.3 addresses higher-level verification properties such as functional correctness.

Chapter 4

The Clustered Multikernel

This thesis contributes the clustered-multikernel design, which we introduce in this chapter. The design was also part of our previous publication [vT12]. The refinement lifting framework we will present in Chapter 5 and Chapter 6 is based on this design.

The chapter is organised as follows: In Section 4.1, we lay out the requirements for our multiprocessor kernel design. In Section 4.2 and Section 4.3, we present the two multiprocessor kernel designs that finally lead to the design of the clustered multikernel in Section 4.4.

For each of the three designs, we discuss its implications on verification, systems design, performance/scalability, and its practical application. We focus on our primary goal: making formal verification tractable. However, we also elaborate on design issues which only impact the systems side.

In Section 4.5, we present the *conversion scheme* to convert a uniprocessor kernel into a clustered multikernel. Along with presenting the scheme, we report on our experience with applying it to seL4. The outcome is *seL4::CMK*, a proof-of-concept implementation of a clustered multikernel based on seL4.

4.1 Design Requirements

Our primary design requirement is that the design we choose must be suitable for formal verification. Recall from Section 1.2 that with a theorem-proving approach, proof complexity strongly depends on program complexity. With regards to concurrency, this means that proof complexity rises rapidly with the complexity of the synchronisation mechanisms used. For example, fine-grained locking and lock-free data structures result in a high number of possible execution scenarios, which all have to be formally identified and covered in the proofs. Similarly, message-based synchronisation results in high proof complexity because invariants about the replicated or partitioned state can be temporarily violated during synchronisation. Furthermore, the message-based synchronisation protocol needs to be proved correct. In summary, we favour a design that results in a low number of possible execution scenarios.

Our secondary design requirements—on the systems side—are partially in conflict with our primary design requirement (verification). Recall from Section 2.8 that when implementing a multiprocessor kernel, we have to make a variety of design decisions with regards to data layout and synchronisation between CPUs. These decisions depend strongly on the kernel's intended area of application.

Remember that the design we are looking for is the design uniprocessor kernels will be converted into. Hence, the refinement lifting framework will be based on this design.

Therefore, we aim for a design that—besides being verification-friendly—is as flexible as possible because we do not know a priori the area of application the converted kernels will be used in. Even when looking at seL4 only, there is a wide range of possible applications because seL4 is a general-purpose microkernel.[1]

Taking the above design requirements into account, we identify the following two designs as the most promising:

- Kernel data is partitioned, but there is no need for the kernel to synchronise the partitioned data. In other words: We have full parallelism, but no sharing or synchronisation of kernel data is necessary. We present such a design in Section 4.2.
- The second design is at the opposite end of the design spectrum: All kernel data is shared, but we use maximally coarse-grained locking, i.e. one single big lock. This dramatically reduces proof complexity because of the small number of possible scenarios that can arise from concurrent execution. We discuss this design in Section 4.3.

4.2 Multikernel Design

In this section, we present the *multikernel* design, originally introduced by Baumann et al. [BBD+09, BPS+09, SPB+08], whose research we summarised in Section 3.1.6.

The main idea behind the multikernel design is to look at the underlying hardware as a distributed system, in which each CPU is treated as a *node*. Each node runs an autonomous kernel instance. As such, nodes do not share any kernel data. However, sharing of memory between nodes is possible at user level. As a consequence, coordination between nodes has to be implemented via *shared memory* at user level. Capability-based kernels such as seL4 can implement shared user-level memory by providing each node with capabilities that point to user-level frames backed by the same physical memory. Optionally, IPIs (inter-processor interrupts) can be used to reduce communication latency between nodes. In this case, the kernel API needs to export IPI handling to user level. The multikernel design is depicted in Figure 4.1.

Normally, OS kernels are self-contained, i.e. they are the minimal TCB (trusted computing base) of the system. In other words: The kernel is able to protect its own integrity, no matter what code is executed at user level. However, the multikernel design does not prescribe this. Moreover, *Barrelfish*, the proof-of-concept multikernel implementation of Baumann et al. [BBD+09], is designed such that the kernels have to trust the user-level *monitors* to work correctly in order to guarantee their own integrity. However, in the area of high-assurance kernels, such a design is problematic. Verified kernels are usually self-contained, and proven to be, in order to be able to guarantee correctness regardless of the code that is run at user level.

For this reason, we aim for a self-contained kernel and restrict the multikernel design accordingly. We derive a *restricted multikernel* design which requires that the kernel must be able to protects its own integrity, independent of what is run at user level. The consequence of this is that—in contrast to Barrelfish—we cannot let a user-level monitor manage dynamic allocation of kernel memory across nodes. Mismanagement could result in the same physical memory region being allocated for different objects on different nodes, most likely resulting in kernel corruption.

[1] Please refer to Section 2.5 for an introduction to seL4.

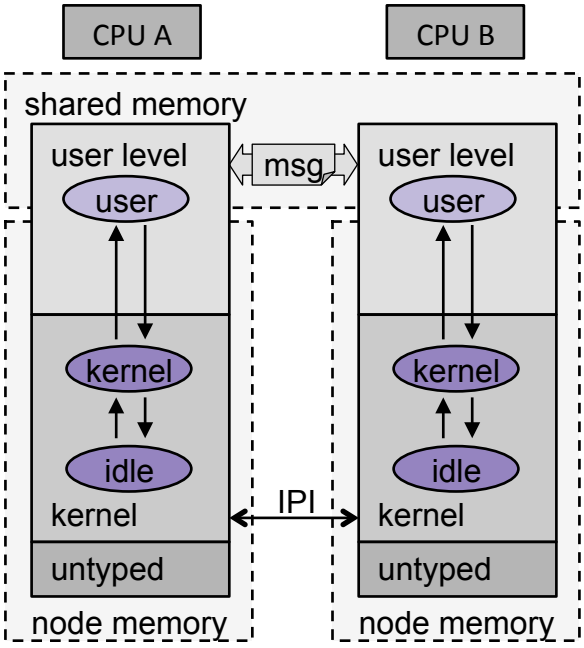

Figure 4.1: *Multikernel Design*

As a consequence, in order to guarantee that no kernel memory is accessed by multiple nodes, the restricted multikernel design statically assigns it to nodes during bootstrapping.

4.2.1 Implications

The restricted multikernel design satisfies our primary design requirement: It is verification-friendly because due to the strict isolation between nodes, there is no concurrent data access in the kernel.

With regards to systems design, the most important implication is that a multikernel API does not export a single-system image to user level. Instead, user level has to deal with a distributed system of independent nodes. Coordination between nodes needs to be implemented at user level via shared memory (and optional IPIs).

For example, a user might wish to execute a classical multi-threaded application across multiple CPUs, which requires an address space to span multiple nodes. Page tables are kernel data and cannot be shared across nodes. Therefore, the said address space has to be created independently but cooperatively on each node. It has to be ensured that on each node, the same virtual address is mapped to the physical address (of shared user-level memory). If the address space is modified at runtime (pages unmapped/remapped), the required coordination between nodes has to be implemented at user level.

The same applies if the user desires load balancing, i.e. if CPUs are dynamically allocated to the application that needs them most. Reassigning CPUs to threads (and address spaces) across nodes requires explicit communication implemented at user level.

Last but not least, the static assignment of kernel memory to nodes in our restricted multikernel design prevents dynamic kernel resource balancing between nodes. Therefore, the system designer has to determine—at boot time—the maximum kernel-memory consumption of each node for the entire time the system will be running. Note, however, that balancing of user-level resources is still possible: Authority over shared user-level memory and memory-mapped devices can be dynamically moved between nodes by cooperating user-level managers.

Nevertheless, the restricted multikernel inherits several advantages from the multikernel: Obviously, there are no locks in the kernel. Therefore, system calls are never blocked and can proceed regardless of what happens on other nodes. Furthermore, the strict assignment of kernel memory to nodes increases kernel data locality. Data locality is crucial for performance and scalability for two reasons:

- If data is accessed by multiple CPUs, it has to be constantly transferred by hardware from one CPU's cache to another's. This behaviour is called *cache-line bouncing*. It can have a dramatic impact on performance and scalability.
- In NUMA (non-uniform memory access) systems, physical memory is partitioned. Each partition is assigned to a cluster of CPUs. Memory accesses within a cluster are much faster than across clusters. For maximum performance, it is therefore necessary to organise data such that it is mostly accessed by CPUs of the same cluster.

Of course, overall system scalability eventually depends on how the remainder of the system is implemented at user level.

4.2.2 Practical Application

Multikernel-based systems have been shown to have competitive and scalable performance [BBD+09, HLSD+13].

Barrelfish

Performance and scalability of entire systems based on the multikernel design have been evaluated by Baumann et al. [BBD+09]. The evaluations were performed on Barrelfish, their multikernel proof-of-concept implementation. We discussed Barrelfish in more detail in Section 3.1.6.

RapiLog

Recently, a restricted-multikernel version of seL4 has been used in the *RapiLog* [HLSD+13] project. The goal of RapiLog is to reduce complexity of DBMSes (database management systems) by rendering unnecessary their logging mechanisms. To ensure durability in case of failures in absence of logging, the DBMS is run in a virtual machine on a formally verified hypervisor, which is guaranteed to never crash. Instead of synchronously logging committed database transactions to disk, the DBMS hands over the data to the hypervisor, which asynchronously writes it to disk eventually. Durability is guaranteed because even if the guest OS or DBMS crashes, the hypervisor is guaranteed not to crash and will eventually write the data to disk. RapiLog also deals with power failures by detecting them and writing the buffered data to disk immediately. This works because most computer hardware continues running for a fraction of a second after power is cut.

RapiLog uses seL4 as the verified hypervisor. However, in order to generate a realistic DBMS workload, the power of multiple CPUs is needed. This requires a multiprocessor guest OS and a multiprocessor hypervisor. The former is not a problem: RapiLog uses Linux. In order to get the latter, RapiLog uses a multikernel version of seL4. In fact, the multikernel version that is used is derived from seL4::CMK, the clustered-multikernel version of seL4 we will describe in Section 4.5.

The RapiLog project showed that the restricted-multikernel version of seL4 can be successfully used as a multiprocessor hypervisor. For the performance evaluation, they used a machine with four cores (and two hardware threads per core). In the virtual machine on top of seL4, they ran Linux as guest OS. The workload inside Linux was a database with 32 clients. They measured a virtualisation overhead of up to 9%. However, only 1.05% of the total CPU time was spent outside the virtual machine. The remaining 8% overhead was attributed to the Intel hardware virtualisation extensions [AA06], or more precisely, to the *extended page table (EPT)* mechanism [VMw09]. This means that the virtualisation overhead caused by the multikernel version of seL4 was only 1.05%.

Conclusion

We believe multikernels to be a practical kernel-architectural approach to system structuring, as demonstrated by the above two systems.

4.3 Big-Lock Design

In the big-lock design, the entire kernel is treated as one single critical section. This means that a single lock protects all kernel data, which is shared between all CPUs. The big-lock design is depicted in Figure 4.2.

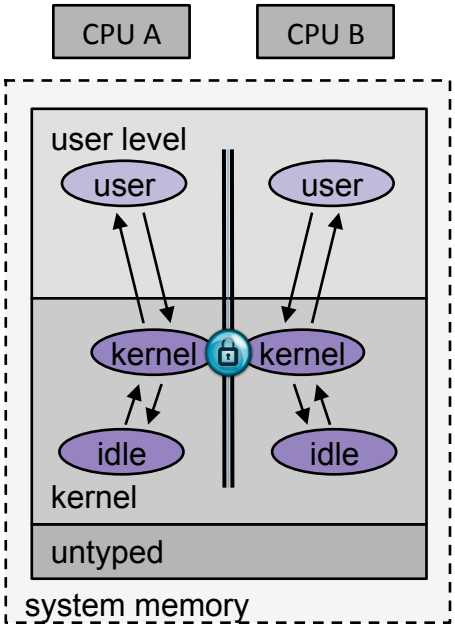

Figure 4.2: *Big-Lock Design*

The advantage of the big-lock design is its simplicity. For this reason, it has been the preferred design for kernels used in security- and safety-critical environments. Examples are QNX (Section 3.1.3) and OKL4 (Section 3.1.4), which are both commercially widely deployed.

4.3.1 Implications

The big-lock design satisfies our primary design requirement: It is verification-friendly because due to the big lock around the entire kernel, there is no concurrent data access in the kernel.

On the systems side, the big-lock design has the advantage that—unlike the multikernel—it inherently exports a single-system image to user level. Multi-threaded applications that have been written for a uniprocessor kernel can be run unmodified on a big-lock version of this kernel and automatically benefit from the power of multiple CPUs at user level.

Furthermore, unlike in the restricted multikernel, resource balancing between CPUs is possible. In fact, it is automatic: Resources are not assigned to CPUs, therefore, they can be used by any CPU any time.

Load balancing of CPUs is similar. If the big-lock kernel implementation uses one runqueue for all CPUs, load balancing is automatic. In any case, implementing per-CPU runqueues would not make much sense in a big-lock design because no scalability can be gained.

However, the obvious drawback of the big-lock design is its poor scalability. In fact, the kernel itself is not scalable at all. Nevertheless, this does not necessarily preclude overall system scalability. We explain this with the help of Amdahl's law.

Definition 4.3.1 (Amdahl's Law)

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

When parallelising a system, the maximum speedup S depends on the degree of parallelism N and on P, which is the fraction of the program code that can be parallelised.

According to the law, scalability of a program directly depends on the fraction of the code that is run in parallel vs. the code that has to be serialised. In case of a program with a single critical section, the fraction can be calculated as the amount of code inside the critical section vs. outside the critical section. A system based on a big-lock kernel is an instance of such a program. The critical section is the kernel, and the code outside the critical section is the code run at user level. Hence, we can conclude that scalability of such a system directly depends on the fraction of CPU time spent in the kernel vs. the time spent at user level.

The actual value of this fraction—and therefore overall scalability—depends on mainly two factors:

- **kernel size**

  The size of the kernel has a major impact on this fraction. For example, in a system with a monolithic kernel such as Linux, much more time is spent in the kernel than in a microkernel-based system, where most OS services are implemented at user level.

- **actual workload**

  There is a strong dependency on the workload currently running on top of the kernel. For example, a computationally intensive scientific application rarely makes system calls or triggers exceptions and would therefore scale well on a big-lock kernel. In contrast, we would see bad scalability for applications that are system-call-intensive, e.g. because they perform a lot of I/O.

In practice, overall scalability of a computer system depends on more than Amdahl's law. A very important additional factor is data locality, which we explained in Section 4.2.1. In a simple big-lock kernel, all kernel data is shared between all CPUs, which results in

bad data locality. There are possibilities to replicate or partition certain parts of the kernel state in order to improve data locality. However, such improvements result in considerable additional kernel complexity which can render verification intractable despite the big-lock design's simplicity.

4.3.2 Practical Application

Due to the simplicity of the big-lock design, it has been used widely in the past for kernels of different types and sizes, e.g. early versions of Linux. However, there are few results published which thoroughly evaluate the scalability of a system based on a big-lock kernel, i.e. to which degree of parallelism such a system scales, depending on the kernel size and the type of workload.

The *OKL4 Microvisor* [Ope] is one of the few big-lock kernels for which a scalability evaluation has been performed [Lyo11]. An interesting finding was that the most intensively used system calls do not need to acquire the lock because they only work on CPU-local data. This is a hint that it might be generally possible to structure a big-lock kernel such that frequently accessed kernel data is CPU-local and accessed from outside the big lock; and that this is possible without incurring substantial additional complexity.

Regarding the remaining system calls, the evaluation found that on the NaviEngine platform, scalability is degraded more by cache contention than the lock itself. In other words, fine-grained locks would not remove the scalability problem. It is more likely that better data locality would.

We presented OKL4 and its evaluation in more detail in Section 3.1.4.

4.4 Clustered-Multikernel Design

We conclude that the two verification-friendly designs we presented above—the restricted multikernel and big-lock kernel—have largely opposite advantages and drawbacks: The former has high scalability while the opposite is true for the latter. However, the former potentially incurs additional complexity due to the distributed-system view at user level whereas the latter provides a simple single-system image.

Remember that we aim for a flexible verification-friendly design. To this end, we propose a configurable combination of the restricted-multikernel and big-lock designs. We call the resulting design a *clustered multikernel*.

The clustered multikernel starts out as a restricted multikernel, but instead of running one CPU per node, multiple CPUs can be *clustered* into (assigned to) a node. Within each node, we apply the big-lock design to synchronise its CPUs. Figure 4.3 depicts a clustered multikernel with two nodes and two CPUs per node.

The clustered multikernel has two main advantages: (1) It is verification-friendly because it is a combination of the restricted-multikernel and big-lock designs. (2) It is flexible because it offers a variety of cluster-configuration options.

From an inverse point of view, the restricted multikernel and the big-lock kernel are nothing else but two distinct configurations of the clustered multikernel: The restricted multikernel is a clustered multikernel where each node runs on exactly one CPU (cluster size is one CPU). The big-lock kernel is a clustered multikernel with one node that spans all CPUs. Therefore, a clustered multikernel can be used whenever a restricted multikernel or big-lock kernel is needed.

In addition, there is a variety of possible configurations by using intermediate cluster sizes. It is explicitly allowed for different nodes to span a different number of CPUs. In

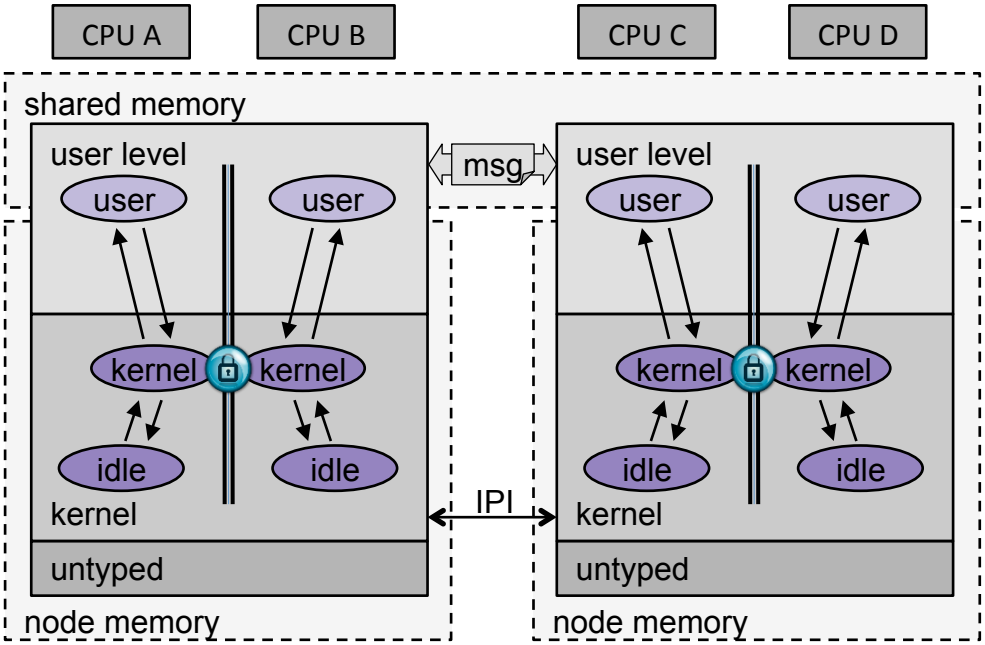

Figure 4.3: *The Clustered Multikernel*

other words, clusters need not be of the same size.

The clustered multikernel offers a configurable trade-off between scalability and flexibility. When building systems on top of it, we can leverage the full scalability between nodes (which we do not have within nodes); and we can leverage the full flexibility of kernel resource balancing within nodes (which we do not have between nodes). By choosing a cluster configuration—i.e. assigning CPUs to nodes—we can fine-tune this trade-off according to our needs.

In Section 7.1.4, we will discuss a possible extension of the clustered-multikernel design that allows balancing of free kernel memory between nodes.

4.4.1 Implications

The primary advantage of the clustered multikernel is that it is a flexible, yet verification-friendly design. Its flexibility allows it to be configured with the underlying hardware in mind.

For example, multiple cores within a processor die can be clustered into a node because potential scalability problems are mitigated by their tight coupling. Similarly, nodes of a clustered multikernel can be aligned with NUMA nodes, which allows NUMA-aware memory assignment to nodes.

Clustering also suits architectures with "islands of cache coherence". Such islands can be aligned with nodes. Within a node, cache coherence allows working with a single-system image. Between nodes, communication channels can be implemented specifically with the reduced coherence guarantees in mind.

The isolation between nodes also allows clustering to be used to draw performance-isolation boundaries for real-time systems in cases where no hardware resources are shared across boundaries. One possibility to achieve this is by drawing the boundaries between NUMA nodes. Another possibility is to draw them between processors and ensure that all code and data is in the processors' local caches. Inside the performance-isolated domains, we can leverage the flexibility of a big-lock kernel.

The clustered multikernel resembles the clustered kernels of the early 90s, Hurricane [UKGS93] and Hive [CRD+95]. Despite many benefits, these kernels suffered from high complexity and performance problems for certain kinds of applications (as discussed in Section 3.1.7). However, these performance evaluations are of limited significance in today's view since they are based on large-scale NUMA systems of that time. While today's multiprocessor systems often have a NUMA architecture as well, the NUMA nodes themselves are very different from back then: Instead of multiple single-threaded processors sharing a memory bus, today's processors consist of dozens of tightly-coupled cores and multiple layers of processor-local caches. Often, a NUMA node consists of a single such processor, thereby removing the need for memory-bus sharing.

Furthermore, we conjecture that the aforementioned problems of these clustered kernels were intensified by trying to hide clustering from user level and provide a single-system image. We believe that this is not necessary for most of the applications you would run on a clustered multikernel today.

4.4.2 Practical Application

A recent example of system design that can benefit from running on top of a clustered multikernel is *Multimed* [SSGA11]. Multimed is a multiprocessor DBMS design aimed at scaling to a large number of CPUs. It follows the idea of Barrelfish (Section 3.1.6) in treating a multiprocessor machine as a distributed system. In contrast to the monolithic designs of most existing DBMSes, Multimed is organised as a system of distributed database engines. Each engine spans a configurable set of CPUs. One engine is the designated *master*. Only the master is allowed to modify the contents of the database. The other engines are called *satellites*. Each satellite has its own replica of the database. When the master makes changes to the database, they are propagated via message-passing to the satellites, which in turn, update their replicas.

Database queries of clients are routed to the satellites by a *dispatcher*, which implements a load-balancing policy. Database updates are directly routed to the master, which can also process queries when it is not busy processing updates.

Figure 4.4 depicts a Multimed system with one master and four satellites, running on a multiprocessor system with eight processors (P0-P7), each having six cores. There are four NUMA nodes, each spanning two processors. The diagram shows that cores can be freely assigned to database engines (master and satellites).

The Multimed design bears a strong resemblance to the clustered multikernel: Both designs are based on clusters that can be freely configured depending to the expected workload and the underlying hardware. The clusters are isolated except for dedicated communication channels. Clusters itself are monolithic, i.e. they provide a single-system image. In case of Multimed, the single-system image is the database instance provided by the DBMS engine.

We strongly suspect that the structural similarities between the Multimed and the clustered-multikernel designs benefit overall system performance and scalability in case a Multimed system is run on top of a clustered multikernel using the same cluster configuration.

4.5 Conversion Scheme and seL4::CMK

In this section, we present our conversion scheme to convert a uniprocessor kernel into a clustered multikernel. Along with presenting the scheme, we report on our experience

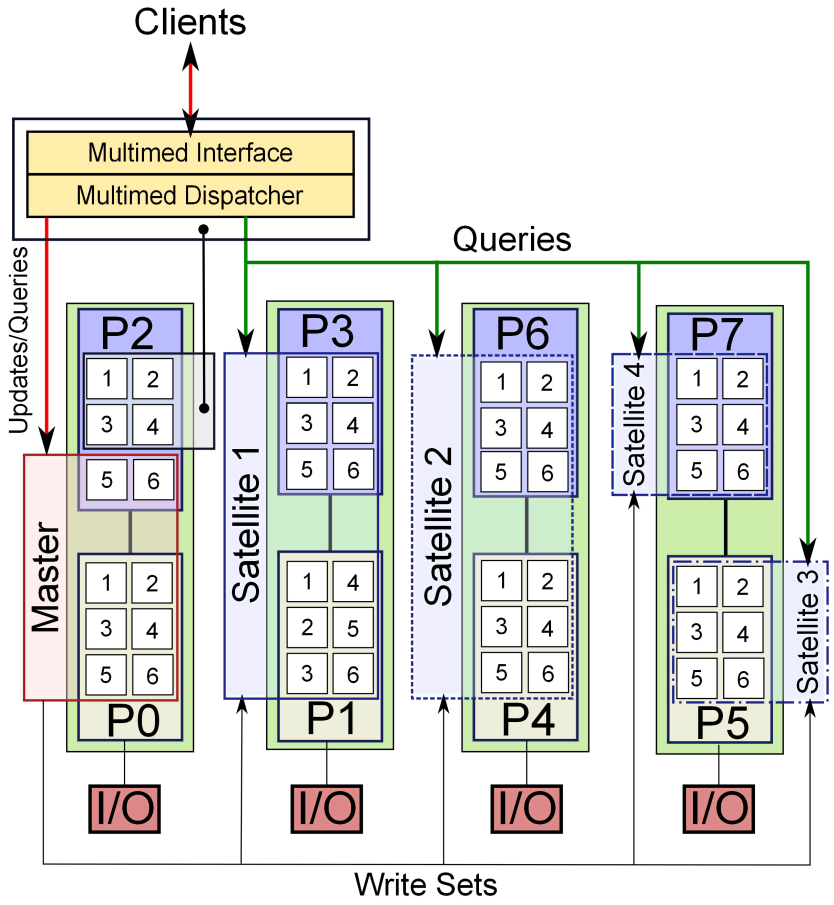

Figure 4.4: *The Multimed Distributed Database Design [SSGA11]*

with applying it to seL4. The outcome is *seL4::CMK*, a proof-of-concept implementation of a clustered multikernel based on seL4.

4.5.1 CPU Architecture

Recall from Section 2.5 that only seL4's ARMv6 version is formally verified. As such, it is the only version of seL4 that can be lifted by our refinement lifting framework. Consequently, the formal application of the refinement lifting framework to seL4 will concentrate on seL4's ARMv6 version.

However, on the implementation side, we decided to work with the x86 version of seL4. This means that seL4::CMK runs on x86 multiprocessor systems. The reasons for this decision are the following: First, when work on this thesis started, ARMv6 multiprocessor systems were practically unavailable, compared to the abundant availability of x86 multiprocessor systems. Second, and more importantly, there are no highly-parallel multiprocessor systems for ARMv6, neither are there NUMA systems. An implementation for ARMv6 would therefore prevent a convincing performance/scalability evaluation of the clustered-multikernel design (Section 7.1.5).

Note, however, that the discrepancy between ARMv6 and x86 is smaller than expected. For example, the majority of seL4's code is architecture-independent. The ARMv6 and x86 versions are even closer on the abstract level. For example, porting the abstract specification that we will present in Section 5.4 from ARMv6 to x86 would only require a dozen LOC of changes, including the required changes to the proofs.

Finally, note that there are plans to also formally verify the x86 uniprocessor version

of seL4, of which an executable intermediate specification has already been written in Haskell.

4.5.2 Bootstrapping

A large part of the changes listed in the conversion scheme below are concerned with kernel bootstrapping. Therefore, we first discuss our bootstrapping approach before presenting the scheme.

When a multiprocessor system is powered up, the initial CPU is started first. It usually executes some platform-specific configuration code (such as a BIOS or EFI) from where a boot loader (e.g. GRUB [GNU]) is started. The boot loader then fetches the *kernel image* from ROM, disk or network and loads it into memory. After that, the kernel's bootstrapping code is called.

In case of a uniprocessor kernel, bootstrapping code now initialises the necessary hardware and kernel data structures and finally starts the initial thread. In a clustered multikernel, however, bootstrapping also has to coordinate starting all remaining CPUs, instructing them what to do and how to initialise themselves.

There are multiple ways to bootstrap a clustered multikernel. Our proposed way is designed such that it is able to encapsulate a mostly unchanged uniprocessor bootstrapping, thereby simplifying the conversion process (of the code and the specification). We bootstrap a clustered multikernel in two parts, as depicted in Figure 4.5: Part 1 of bootstrapping is responsible for discovering available resources (CPUs, memory, devices), initialising the platform (IRQ controllers, timers, PCI bus, etc.), writing down the cluster configuration (CPU/memory assignment to nodes) and starting the remaining CPUs. Each node's first CPU reads the cluster configuration and then calls part 2 of bootstrapping, passing the node's configuration as function-call arguments. As such, part 2 now has a completely node-local view, which allows it to be an almost unchanged version of the uniprocessor bootstrapping.

4.5.3 Conversion Scheme

We are now ready to present the scheme to convert a uniprocessor kernel into a clustered multikernel. At a high-level, the scheme consists of the following steps:

1. divide up the bootstrapping code into three fragments:
   initialisation of (1) platform, (2) CPU, (3) data structures
2. write the code for bootstrapping part 1
3. write the code for bootstrapping part 2
   (by almost entirely reusing the uniprocessor bootstrapping code)
4. implement the big lock
5. solve the problem of CPU identity within a node
6. implement remote TLB invalidation
7. solve what we call the *running-thread problem*
8. optionally implement asynchronous inter-node signalling

We explain each step in the following list, which is an expansion of the list above. Where necessary, we refer to separate sections for more detailed explanations.

1. First, the original kernel's bootstrapping code has to be divided up into three fragments:

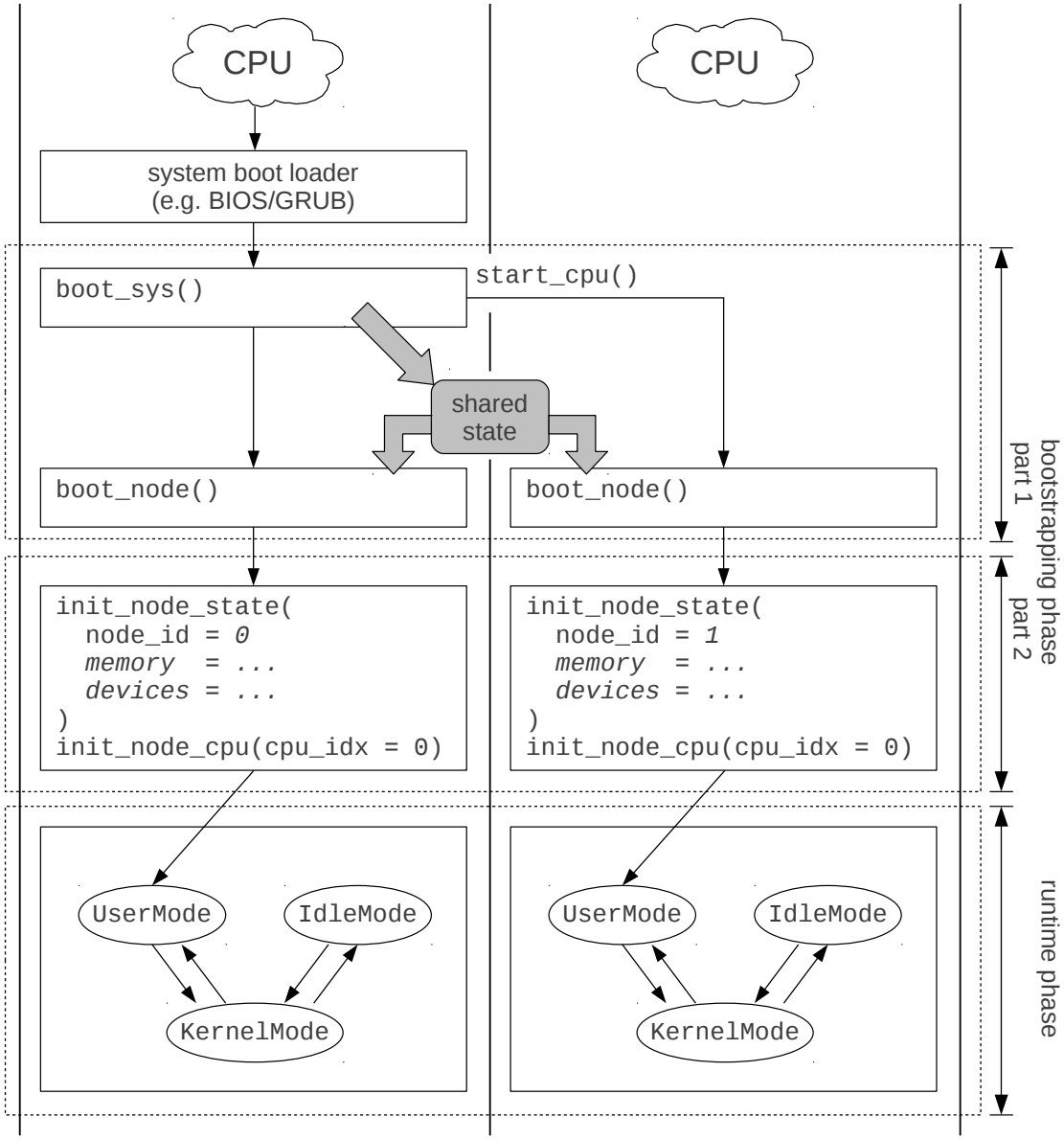

Figure 4.5: *Bootstrapping and Runtime Phases of seL4::CMK (two nodes with one CPU each are depicted)*

- code that discovers and initialises platform devices (IRQ controllers, timers, PCI bus, etc.)
- code that initialises the CPU (configuration registers, on-CPU timers, virtual-memory configuration, etc.)
- code that initialises kernel data structures

In a uniprocessor kernel, it is not necessary to distinguish these fragments because all initialisations have to be done only once. In a clustered multikernel, however, they need to be separated: Kernel data structures need to be initialised for each node, i.e. the fragment needs to be executed once for each node. CPU-specific initialisations need to be done for each CPU, so the fragment needs to be executed once for each CPU. Platform devices exist only once, which means that this fragment needs to be executed once, like for the uniprocessor version.

Luckily, there is a high chance that these fragments are mostly separated already, simply because it is good programming style. This was the case in seL4. Completely separating the fragments only required a few LOC to be changed.

2. Now, part 1 of clustered-multikernel bootstrapping has to be written (`boot_sys()` in Figure 4.5). The implementation starts with the first fragment separated in the first step of the scheme. Following this fragment, new code has to be written that does the following. Remember that at that time, there is only one CPU running.

   - It discovers the available remaining CPUs (which are not started yet).
   - It reads the desired configuration that the user passed via boot loader. The configuration includes the assignment of CPUs and memory to nodes. This configuration is written to a designated region of memory (named *shared state* in Figure 4.5).
   - By definition, the already running CPU is the first node's first CPU. It now starts each other node's first CPU (`start_cpu()` in Figure 4.5).
   - Every node has its first CPU running now (`boot_node()` in Figure 4.5). Each of them reads the configuration from the designated region of memory, concurrently with each other.
   - In each node, the main function of bootstrapping part 2 (`init_node_state()` in Figure 4.5) is called and the configuration passed to it as normal function-call arguments. This is required because here, we transition from a global system view to a node-local view. The only way for information to be passed on is via function-call arguments since for verification reasons, `init_node_state()` cannot access any variables outside the node it is initialising.
   - When bootstrapping part 2 returns, we know that a node's kernel data structures are initialised. Now we can initialise the node's first CPU (by calling `init_node_cpu()`, Figure 4.5) and start/initialise its remaining CPUs.[2]

     The alternative way of starting the remaining CPUs of all nodes by the system's first CPU after all nodes have been initialised it not an option in our case. It requires this CPU to spin-wait until all nodes have been initialised before starting the nodes' remaining CPUs. As we will see in Chapter 5, this is not currently supported by our verification framework.

     Note that the code doing the CPU initialisation is the third fragment separated in the first step of the scheme.

   Implementing bootstrapping part 1 of seL4::CMK required 430 LOC. Writing the abstract specification required a similar effort, which we will discuss in Section 5.4.1.

3. This step is about bootstrapping part 2 (`init_node_state()` in Figure 4.5). As mentioned before, we can almost entirely reuse the third fragment separated in the first step of the scheme (uniprocessor kernel data initialisation). The reason is that we now have a completely node-local view without concurrent data access. The following small modifications have to be made:

   - It has to be ensured that all static global kernel data of the uniprocessor version is replicated for each node and accessed locally within each node. While this

[2] This is not visible in Figure 4.5 because it only depicts nodes with one CPU.

sounds simple at first sight, there are multiple possible ways to implement this, while each way has its own tradeoffs. We will elaborate on this further in Section 4.5.4.

- There will most likely be a global variable in the uniprocessor kernel that contains the pointer to the currently running thread. This variable has to be duplicated for each CPU in a node, which is usually achieved with making the variable an array of the same type. The same applies to other CPU-local data (if there is more).
- The aforementioned configuration also needs to include the physical memory location and size of the region of *shared memory*.[3] The bootstrapping code has to be extended to create user-level frames that cover this region.

When applying this step to seL4, only a dozen LOC of the uniprocessor bootstrapping code had to be changed. The remainder of the code could be used unchanged as bootstrapping part 2 of seL4::CMK. The formal side of this step (abstract specification) will be discussed in Section 5.4.2.

4. The remaining steps are concerned with the runtime phase of the kernel. This step is about inserting the big lock to synchronise access of a node's CPUs to the node's kernel data. The lock can be a normal spin lock. However, we suggest using an MCS lock [MCS91] for the following reason.

   Boyd-Wickizer et al. [BWKMZ12] investigated scalability of Linux and found that non-scalable locks (such as normal spin locks) can have dramatic effects on overall performance, even if only used for very short critical sections. This can be prevented with using scalable locks, such as MCS locks. They also found that scalable locks other than MCS had no significant advantage over MCS locks.

   Lock acquisition/release has to be carefully placed. They will most likely be located in the assembly code that manages kernel entry/exit. The lock has to be acquired after kernel entry and before any of the node's kernel data is accessed. Similarly, it has to be released after the last access and before returning to user level. Outside the lock, only CPU-local data can be accessed (e.g. stack contents).

   In case of seL4::CMK, placement of lock acquisition/release was straight-forward. It had to be placed in the assembly code around kernel entry/exit, which means that the entire C code of the kernel lies within the lock.

5. This step of the scheme is about CPU identity. When entering a clustered multikernel from user level, an entering CPU needs to know its identity in order to be able to know which thread it is currently running. There are multiple ways of achieving this, depending on the architecture in use. On x86, for example, there is the possibility to read the CPU ID from a specific physical memory address. However, this access is relatively slow.

   In seL4::CMK, we use the following trick: Before returning to user level, the CPU stores the pointer to its currently running thread in the *kernel stack* field of its TSS (task state segment). This means that upon kernel entry, the CPU automatically loads its kernel-stack-pointer register (ESP) with the pointer to the currently running

[3] This is the "shared memory" mentioned in Section 4.2, which will end up as user-level memory that is shared between nodes.

thread. This bypasses the need of the CPU to query its ID in order to find the currently running thread.

A second issue with CPU identity is *lazy FPU switching*, which we discuss in more detail in Section 4.5.5.

6. In a multiprocessor system, each CPU has its own TLB (translation lookaside buffer). This means that if two or more CPUs work on the same address space, changes to that address space need to be synchronised between the affected CPUs. Creation of new mappings is not a problem because, if not present in a TLB, they are loaded from the shared page tables. Nevertheless, unmapping pages is critical. If the unmap operation is supposed to take effect immediately, there needs to be a mechanism that allows one CPU to signal another CPU to remove mappings from its TLB and get back an acknowledgement in order to be able to signal to the user that the unmap operation is done. This mechanism is commonly known as *remote TLB invalidation*. Some multiprocessor architectures provide a hardware mechanism to do remote TLB invalidation (e.g. `tlbsync` on PowerPC [IBM05]). On x86, however, the OS kernel needs to implement it.

   In a clustered multikernel, remote TLB invalidation across nodes is not necessary because nodes do not share address spaces. However, we need to implement remote TLB invalidation within a node. We implement a straight-forward scheme in which a TLB invalidation message is broadcast via IPI (inter-processor interrupt) to all other CPUs within the same node whenever a page is unmapped or an address space is destroyed.

   On the specification level, we model broadcasting the TLB invalidation message as an *abstract machine operation*.[4] The addition of this operation to seL4's abstract and intermediate specifications required 60 LOC. Fixing up the refinement proof between these specification levels required 190 LOC.

7. In case the API of the uniprocessor kernel allows threads to directly manipulate other threads at any time, additional care has to be taken to synchronise such operations between a node's CPUs. We call this the *running-thread problem* and will elaborate on it in detail in Section 6.4.

8. This last step of the scheme is optional. It is about implementing inter-node signalling. Recall from Section 4.2 that providing a way of asynchronous signalling between nodes can improve coordination latency—and therefore overall system performance—considerably. Therefore, the goal of this step is to provide a way of sending and receiving such signals at user level. We discuss this step in detail in Section 4.5.6.

This concludes the scheme to convert a uniprocessor kernel into a clustered multikernel.

4.5.4 Static Node-Local Kernel Data

A kernel normally requires both statically and dynamically allocated data structures. In a uniprocessor kernel, statically allocated data structures are usually declared as global

[4] An *abstract machine operation* is a mechanism that seL4 already uses for modelling hardware operations such as local TLB invalidations. It is abstract in the sense that it does not model the impact of the hardware operation on the system state. However, refinement ensures that whenever the abstract specification invokes such an operation, it is also invoked by the implementation.

variables. In seL4, this includes for example the pointer to the currently running thread and to the idle thread, the x86 TSS (task state segment) and descriptor tables (GDT, IDT). When converting into a clustered multikernel, these global variables need to be replicated per node. This can be done in multiple ways, each having a different impact on verification and performance.

Hereafter, we list and discuss the alternatives we considered for seL4::CMK:

- **use an array for each global variable**

  The cleanest solution is to turn every global variable into an array of itself. Each array index corresponds to a node. This has the advantage of being architecture-independent and of the compiler handling the allocation of the necessary memory. However, the variables have to be accessed differently than before. On every access to a global variable, the node index has to be provided. Depending on the architecture, it might be possible to either store it in a CPU register, or to infer it from a CPU register or from the CPU ID. Unfortunately, querying the CPU ID is a relatively slow operation on x86, and most probably also on other architectures. Furthermore, every location in the code from which a global variable is accessed has to be modified, although this can be avoided by using macros.

  As an alternative, all global variables can be moved into a struct, of which an array is declared. This better encapsulates the global variables, which might benefit the approach of creating macros in order not having to make extensive changes to the existing code.

  The impact on (re-)verification of modified code depends on whether the above mentioned macros are transparent to the verification framework, and if not, whether the framework's automation can handle the additional array lookup without user intervention. For seL4, this would not be the case, i.e. each global-variable access would require manual intervention in order to re-establish proofs.

- **use a segment register** (x86 only)

  On x86, there is an additional possibility by using a segment register. For example, the FS register is normally not used by user-level code. To implement this solution, the FS register is loaded with the descriptor of a segment that points to the node's static kernel data, while different segments are assigned to different nodes. The FS register itself is never changed. It therefore permanently stores the assignment of the CPU to a node.

  The advantage of this solution is that it does not require finding the node index on every access. Hence, static kernel data access stays as fast as before. Nevertheless, the refactoring effort is considerable. Access to memory based on the FS register requires assembly code. Each global variable therefore requires an assembly accessor function pair (get/set).

  If there are a lot of global variables, we can use the same alternative as above: encapsulating all global variables into a struct, which reduces the number of required assembly accessor functions to two. However, in order to re-establish proofs, each global-variable access would require manual intervention.

  Note that Linux is using this approach for accessing CPU-local kernel data (checked version: 3.7.3).

- **use virtual memory**

  It is also possible to use paged virtual memory in order to access different nodes' static kernel data. In the solution presented here, all global variables are assigned to a specific compiler output section to which a specific virtual address is assigned. At bootstrapping, each node is allocated some physical memory to hold these global variables. For each node, the aforementioned virtual address is mapped to another physical address, i.e. the address of the physical memory that has been allocated.

  The advantage of this solution is that it is architecture-independent and does not require any node-index lookup. As such, global variable access experiences no slowdown. Furthermore, no changes to the C code are necessary because all global variables stay proper global variables. This means that (re-)verification is trivial.

  The drawback of this solution is that it requires explicit allocation of physical memory for the nodes' static kernel data, thus increasing the complexity of early bootstrapping slightly. It also requires advanced tuning of the linker script, which can be a delicate undertaking.

For seL4::CMK, we decided to implement the last solution because it has the least impact on (re-)verification and incurs no performance degradation. While the second solution incurs no performance degradation either, we excluded it because of its impact on (re-)verification since it either requires countless assembly accessor functions or refactoring all global variables into a struct. Furthermore, both FS and GS registers are already in use by seL4's user-level library in order to store thread-local data. Hence, using the second solution would have required modifying the user-level library not to use the FS or GS registers.

4.5.5 Lazy FPU Switching

Lazily switching the FPU (floating-point unit) of a CPU reduces thread-switch time if there are threads that do not use the FPU (which is a common case). Normally, the FPU registers of a CPU would have to be saved and restored on every thread switch. If lazy FPU switching is implemented, the FPU registers are not saved/restored when switching from/to threads that have never used the FPU. Furthermore, when switching away from a thread that is using the FPU and to a thread that has never used the FPU, the FPU registers are not saved either. The registers are only saved when a thread is switched to that uses the FPU as well. If this thread happens to be the thread that last used the FPU, the registers already contain the required values and no restore has to be performed. In order to prevent illegal access to FPU registers of other threads, the FPU is disabled when running a thread that has never used it. The FPU is enabled for a particular thread as soon as it is used first.

Lazy FPU switching as explained above works if threads always execute on the same CPU, e.g. in a uniprocessor environment or with threads pinned to specific CPUs. In a clustered multikernel, it would only work with a cluster size of one, i.e. in nodes running on one CPU. In multiple-CPU nodes, threads might migrate between CPUs. Therefore, if a thread uses the FPU, its FPU registers have to be saved every time it is switched away from. Otherwise, the thread might end up running on an other CPU later while its FPU register data is still stored in the old CPU.

Consequently, we have to address this issue in case the kernel to be converted uses lazy FPU switching.

For seL4, this is the case. Hence, our solution for seL4::CMK is to change lazy FPU switching to *selective FPU switching*. This means that whenever we switch to a thread that is using the FPU, we restore the FPU registers, and whenever we switch away from it, we save them. In contrast, switching from/to threads that have never used the FPU does not trigger any save/restore. This results in slightly more saves/restores than with lazy FPU switching. However, it is still much more efficient than unconditionally saving/restoring the FPU registers on every thread switch. Changing the FPU switching algorithm involved changing only a handful of LOC.

4.5.6 Inter-Node Signalling

In multiprocessor systems, asynchronous notifications between CPUs are done with IPIs. Hence, if signalling between nodes is desired, it has to be implemented via IPIs. To this end, the kernel has to implement sending and receiving IPIs and provide an abstracted view of this functionality to user level. Receiving is straight-forward, as on most architectures, it is no different from receiving any other IRQ, something that is already implemented by a uniprocessor kernel. On the sending side, however, new code is probably necessary. The IPI sending mechanism is most likely a new system call. However, the exact details depend on the existing kernel API.

In seL4::CMK, we opted for implementing IPI-based inter-node signalling. On the receiving side, nothing had to be changed as the IRQ receiving mechanism could be reused without modification. Specifically, in order to receive an inter-node IPI at user level, an asynchronous endpoint can be registered for a predefined IRQ number. In order to send an IPI, we added a new capability type with a single method called `SendIPI()`. This method takes as arguments the destination node ID and the desired IRQ number to be generated in the destination node. A capability of this new type is given to each node's initial thread.

The initial threads can copy this capability to all entities that require sending inter-node notifications. Entities that are not given a copy of this capability cannot send IPIs. However, with this capability model, it is not possible to disseminate finer-grained authority, e.g. to send IPIs only to a specific node. This restriction can be circumvented at user level by only giving an IPI capability to a trusted manager. Other entities have to send IPIs via this manager, which decides who is allowed to send IPIs to which nodes. Nevertheless, this workaround has a potential performance penalty. If necessary, the restriction could also be removed at the kernel level (without performance penalty) by re-implementing the IPI capability to carry more fine-grained authority information.

The additional capability type required adding about a dozen LOC to the implementation. On the specification level, we model sending IPIs as an abstract machine operation. The addition of this operation and of the new IPI capability type to seL4's abstract and intermediate specifications required 80 LOC. Fixing up the refinement proof between these specification levels required 170 LOC.

Chapter 5

Refinement Lifting Framework: Bootstrapping Phase

In Chapter 4, we have introduced the clustered multikernel and discussed its advantages and limitations from a systems point of view. We also presented the conversion scheme to convert a uniprocessor kernel into a clustered multikernel.

In Chapter 5 and Chapter 6, we focus on the formal aspects. Specifically, we show how the clustered multikernel enables a refinement proof of a multiprocessor kernel. Our goal is to leverage an existing uniprocessor kernel with existing refinement proof and reuse as much as possible from this code and proof base. The process of reusing an existing proof in a new context by adapting or extending it is called *lifting*. For example, a theorem about a kernel-internal function can be reused in a multiprocessor context if we prove that no concurrency is introduced for that particular function.

The refinement lifting framework does this by exploiting the clustered multikernel's design features, specifically the confined concurrent data access. This allows it to lift most of the uniprocessor kernel's refinement proof such that it applies to the clustered-multikernel version. The support for handling the confined concurrency is added to the existing verification framework in a non-intrusive way. The refinement lifting framework accounts for weak memory ordering exhibited by total-store-order (TSO) multiprocessor architectures.

In summary, we aim to convert a uniprocessor kernel into a clustered multikernel and lift its refinement proof with the least engineering effort possible. Our approach retains the formal guarantees of the uniprocessor kernel. However, we make a few additional assumptions in our models, and the original formal guarantees are retained only under these assumptions.

We focus on the seL4 microkernel [KEH+09]. Nevertheless, the results are applicable to other kernels as well if they meet certain conditions. We will discuss these conditions where relevant.

The refinement lifting framework requires a list of theorems to be proved in order to lift the uniprocessor refinement proof into a clustered-multikernel context. To assist with the latter and to reduce proof-engineering effort, we provide a number of proved theorems which are clustered-multikernel-specific but kernel-agnostic. These particular theorems need only be proved on the abstract level because we leverage the fact that the existing refinement proof transfers them down to the concrete level.

We report on our experience with applying the refinement lifting framework to the seL4 microkernel. We apply the framework to seL4's first refinement step (between the abstract and the intermediate specification) as a proof of concept. While we elaborate

on what is involved in also applying it to the second refinement step (down to the C implementation), the formal application to this step is left for future work (Section 7.1.2).

Besides refinement, the framework is also able to lift additional theorems such as invariants that are not required by the original refinement proof. For example, a proved invariant in seL4 ensures that kernel-accessed pages are correctly mapped. This invariant was not necessary to prove refinement. It was proved because it is desirable to know that it holds. Invariants like this can be lifted as well.

Challenges The refinement lifting framework faces multiple challenges: (1) The verification framework used to prove a uniprocessor version of a kernel is likely tailored or even restricted to sequential reasoning, i.e. there is no notion of concurrency in the basic model. This is specifically true for seL4's verification framework. Hence, we need to provide a non-intrusive way of augmenting such a framework with limited support for concurrency. (2) In order to reduce proof complexity introduced by having to reason about concurrency, we need to structure the new proof in a way that allows most parts of the kernel to be reasoned about sequentially. Having to reason about concurrent actions should be reduced to the few spots where concurrency is unavoidable in a clustered multikernel. (3) The framework extension and proof structure should allow reusing the uniprocessor proof with only minimal modifications.

Presentation We present the refinement lifting framework in two chapters: Chapter 5 covers the bootstrapping phase of the kernel while Chapter 6 handles the runtime phase. Assumptions and limitations are mentioned throughout these chapters and are discussed in detail at the end of each chapter.

The basic ideas behind the refinement lifting framework have already been published in our previous work [vT10, vT12].

5.1 Chapter Overview

In Chapter 5, we present the part of the refinement lifting framework that is concerned with the bootstrapping phase of the kernel.

Multiprocessor Execution Model Recall from Section 4.5 that in the bootstrapping phase, we have concurrent access[1] to shared data. Therefore, we augment the existing verification framework with a *multiprocessor execution model*, which we introduce in Section 5.2. The model accounts for weak memory ordering exhibited by total-store-order (TSO) multiprocessor architectures.

Sequential Semantics The type of state monads described in Section 2.3 and used in the specifications of seL4 and seL4::CMK assumes that during the monadic execution, we observe *sequential semantics*. This means that the values stored in variables do not change unless we change them. In other words: All reads observe the last write.

As a first step towards proving this assumption, Section 5.3 contributes the kernel-agnostic *sequential-semantics theorem*, which uses the TSO model to prove that in a bootstrapping situation, every CPU observes sequential semantics, under the assumption that memory fences are used correctly and a specific memory-access pattern is adhered to.

[1] We define a memory *access* to be either a read or a write access.

These assumptions have to be proved for the specification in question when applying this theorem. The proof of this theorem is deferred to the end of this chapter (Section 5.12.1).

Abstract Specification Section 5.4 is devoted to formally specifying the code of a clustered multikernel's bootstrapping phase. First, we present our approach to connecting a bootstrapping specification to the system boot loader. Second, we show how a specification dynamically allocates and accesses data structures. Third, we explain how we connect a specification to the multiprocessor execution model. We do all of this with the example of an abstract specification of seL4::CMK's bootstrapping, which we provide.

Bootstrapping Correctness In Section 5.5, we combine this specification with the sequential-semantics theorem. The result is the *kernel-memory-sequential-access theorem*, which states that during bootstrapping of seL4::CMK, each CPU observes sequential semantics. This was the assumption we made by using a non-deterministic state monad for seL4::CMK's abstract specification. We therefore prove that this assumption is correct.

The kernel-memory-sequential-access theorem is proved by applying the sequential-semantics theorem and proving its assumptions, i.e. that memory fences in seL4::CMK's specification are placed correctly and that the specification adheres to the memory-access pattern mentioned above. The proof of these assumptions—-and therefore the kernel-memory-sequential-access theorem—is deferred to the end of this chapter (Section 5.12.2).

At this point, Section 5.6 takes a break with an intermediate conclusion of what we have presented so far and what lies ahead. We continue with concepts, techniques and theorems that—while still talking about bootstrapping—are required by the runtime phase of the kernel.

Node Isolation An important design aspect of the clustered multikernel is that a node's kernel is isolated from other nodes. This makes verification more tractable because we can reason about each node in isolation. It is important for part 2 of the bootstrapping phase, and even more, for the runtime phase. To this end, Section 5.7 introduces the *kernel isolation theorem*, which states that the nodes' kernels are bootstrapped isolated.

In Section 5.8, we present the *user-level isolation theorem* and the *user-level sharing theorem*. They assert useful properties about user-level memory, which a user-level application can rely on. For example, together with the kernel isolation theorem, the user-level isolation theorem provides node-isolation guarantees which can be useful for building systems on top of seL4::CMK. Nevertheless, the user-level theorems are not strictly necessary for refinement lifting.

We defer the proofs of these theorems to the end of this chapter (Section 5.12).

Connection to L4.verified In Section 5.9 we introduce a technique to connect seL4::CMK's abstract bootstrapping specification with the runtime-phase specification of seL4 (from the L4.verified project). This is necessary because the two specifications are based on slightly different formalisations.

Refinement Theorem Finally, in Section 5.10, we present the *multikernel refinement theorem*. This theorem shows refinement of a multikernel-version of seL4, i.e. a configuration of seL4::CMK where each node runs on exactly one CPU. The step to a *clustered multikernel*—i.e. multiple CPUs per node—is the topic of the next chapter (Chapter 6).

Limitations In Section 5.11, we discuss limitations and the assumptions we make for the refinement lifting of the bootstrapping phase.

Presentation Presentation and discussion of the aforementioned proofs is deferred to the end of this chapter (Section 5.12), which we conclude in Section 5.13.

Note that our work-in-progress paper [vT10] already covered the basic idea and an early stage of the research presented in this chapter.

5.2 Multiprocessor Execution Model

In this section, we introduce our multiprocessor execution model, which augments the existing verification framework with support for concurrency.

5.2.1 Challenges

When modelling concurrent execution and data access, it is important not to oversimplify the source of concurrency. For example, modelling concurrent data accesses as interleavings of sequential accesses is only applicable if the accesses are atomic and modified data is immediately visible to all potential readers.

The source of concurrency in our case is the hardware, i.e. multiple CPUs running concurrently and potentially accessing the same memory addresses. Unfortunately, the interleaving model is not applicable to most of today's multiprocessor architectures because they exhibit weak memory ordering (see Section 2.7). As a consequence, our multiprocessor execution model needs to correctly model such a weak-memory-ordering architecture. Modelling concurrent data access as interleaving of sequential accesses is insufficient.

5.2.2 Formal TSO Model

To this end, we contribute a formal TSO[2] model. The basics of this model are published in our work-in-progress paper [vT10]. However, at that time, the model did not yet cover weak memory ordering.

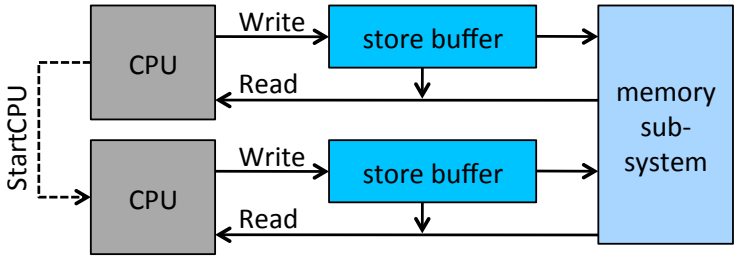

Figure 5.1: *TSO Model*

Our TSO model is depicted in Figure 5.1 with the example of two CPUs. It is formalised with operational semantics,[3] i.e. it explicitly models the CPUs, their store buffers and the memory subsystem. It is similar to the TSO models presented in Section 3.3.3 and

[2]TSO memory ordering is explained in Section 2.7.1.
[3]Operational semantics are explained in Section 3.3.3.

Section 3.3.4. Its novelty is that it explicitly models CPUs starting other CPUs in presence of weak memory ordering.

The modelled interactions (arrows in the figure) are: (1) a CPU reading from the memory subsystem, (2) a CPU writing to its store buffer, (3) a store buffer writing to the memory subsystem, (4) a CPU reading directly from the store buffer (store-buffer forwarding), and (5) a CPU starting a new CPU.

On a multiprocessor system, each CPU runs its own *instruction stream* independently from the others. There is no common clock, and as such, it cannot be predicted in which global order instructions of different CPUs are executed. On the other hand, modelling a global order of single instructions is not necessary anyway because instruction streams only interact with each other when reading/writing memory. Between two instructions that access memory, a CPU runs independently from all others. This allows us to abstract away from native CPU instructions and only care about *abstract instructions*. An abstract instruction models all native CPU instructions that interact with the memory subsystem in the same way. For example, the abstract read instruction models all native CPU instructions that read from memory etc.

An *abstract parallel program* consists of abstract instructions. The TSO model non-deterministically and symbolically runs such a parallel program and computes all possible resulting *memory-access scenarios*.

The TSO model can be used to prove presence or absence of memory-access scenarios resulting from symbolically running a given abstract parallel program. Note that a program definition is not needed. The program can be arbitrarily underspecified. However, heavier underspecification of the program also means that we can prove less detailed properties about the resulting memory-access scenarios.

We start with defining the basic types of the TSO model: CPU ID, physical memory address, and physical memory address region.

Definition 5.2.1 (Basic Types)

```
types cpu_id_t    = nat
types paddr_t     = nat
types p_region_t  = paddr_t ⇒ bool
```

We define the following four abstract instructions.

Definition 5.2.2 (Abstract Instructions)

```
datatype instr_t =  InstrRead p_region_t
                  | InstrWrite p_region_t
                  | InstrMFENCE
                  | InstrStartCPU cpu_id_t (instr_t list)
```

InstrRead and InstrWrite model all native CPU instructions that read or write memory. Native CPU instructions between them (in an instruction stream) are CPU-local (e.g. register operations) and have no effect on memory. They merely affect the parameters of the next abstract instruction, which then interacts with the memory subsystem. The `p_region_t` datatype content specifies the set of physical memory addresses that is read/written atomically by the specific instruction. The model supports any combination of memory addresses to be accessed atomically. However, a sensible specification would only use sets that conform to the *unit of atomicity* of the modelled architecture. For example, for 32-bit architectures, the set would contain four consecutive and aligned memory addresses. Nevertheless, the user of the TSO model is free to specify whatever they want.

InstrMFENCE specifies a *memory fence*, which forces the store buffer to be emptied before any further instruction is executed.[4] InstrStartCPU instructs the CPU to start a new CPU with ID `cpu_id_t`. The new CPU will start executing the abstract parallel program `instr_t list`. Note that the model does not limit the maximum number of CPUs.

While symbolically executing the parallel program, the TSO model accumulates the non-deterministic set of possible memory-access scenarios, each of which is represented as a list of memory accesses. Figure 5.2 depicts the input/output of the TSO model. On the left side, a typical bootstrapping abstract parallel program is shown where we have one CPU initially running the program that instructs that CPU to start other CPUs. This results in a tree where each InstrStartCPU creates a new branch.

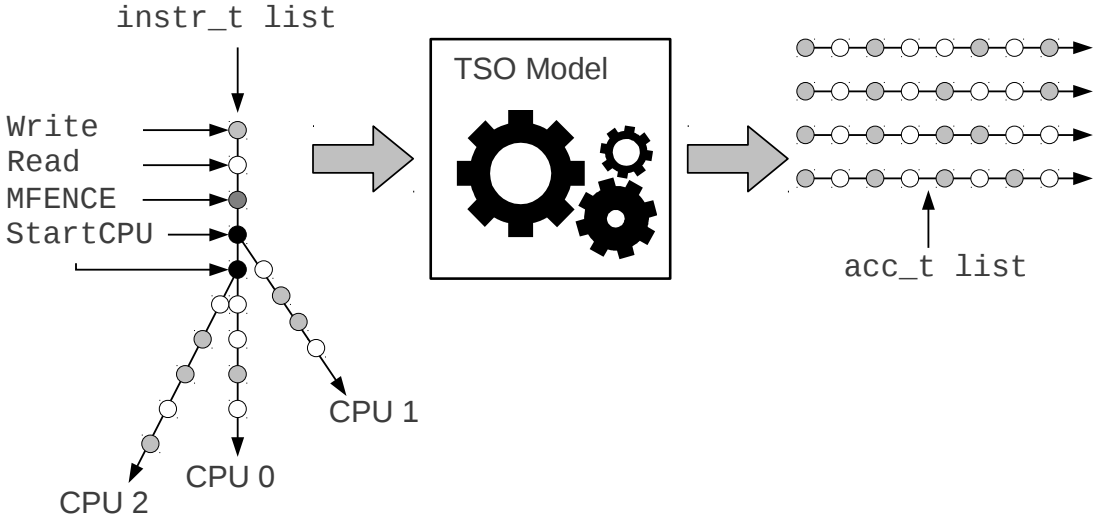

Figure 5.2: *Input and Output of the TSO Model*

Definition 5.2.3 (Memory Access)

datatype `acc_t` = AccRead `p_region_t cpu_id_t` | AccWrite `p_region_t cpu_id_t`

A memory access can either be a read or a write access. In both cases, it is specified which memory addresses were accessed (atomically) and by which CPU. On the right hand side of Figure 5.2, the non-deterministic set of all possible memory-access scenarios is depicted.

Throughout the execution, the TSO model maintains a *system state* which includes the parallel program remaining to be executed, the contents of the store buffers and the already accumulated current memory-access scenario. We operate with a set of system states that account for the non-determinism introduced by multiple CPUs and multiple store buffers.

Definition 5.2.4 (System State)

types `cr_t` = `cpu_id_t ⇒ bool`
types `cil_t` = `cpu_id_t ⇒ instr_t list`
types `csb_t` = `cpu_id_t ⇒ p_region_t list`

[4] According to Sewell et al. [SSO+10], flushing the store buffer correctly models a memory fence on a TSO architecture.

```
record mpss_t =  mpss_cr   :: cr_t
                 mpss_cil  :: cil_t
                 mpss_csb  :: csb_t
                 mpss_al   :: acc_t list
```

The *mpss_t* (**m**ulti**p**rocessor **s**ystem **s**tate) consists of the accumulated memory-access scenario mpss_al and a collection of per-CPU states: mpss_cr (**C**PU **r**unning) stores which CPUs are running, i.e. have already been started; mpss_cil (per-**C**PU **i**nstruction **l**ist) stores the program to be executed by each CPU, and mpss_csb (per-**C**PU **s**tore **b**uffer) stores each CPU's store-buffer content.

The operation of the TSO system is modelled by a *transition relation* trans_rel that is defined by *transition rules*. These rules specify valid transitions between *system states*.

Before we define the transition rules themselves, we first define a collection of *predicate* and *modifier* helper functions.

For handling *instruction lists* (abstract parallel programs), we define the following helper functions.

Definition 5.2.5 (Instruction-List Helper Functions)

```
cil_next :: instr_t ⇒ cpu_id_t ⇒ mpss_t ⇒ bool
cil_next i c s ≡ ∃il. i · il = mpss_cil s c

cil_dequeue :: cpu_id_t ⇒ mpss_t ⇒ mpss_t
cil_dequeue c s ≡ s(|mpss_cil := (mpss_cil s)(c := tl (mpss_cil s c))|)
```

With the predicate cil_next we can test if i is the next instruction that CPU c will execute. The modifier cil_dequeue removes the next instruction from the instruction list of CPU c.

For handling store buffers, we define the following helper functions.

Definition 5.2.6 (Store-Buffer Helper Functions)

```
csb_next :: p_region_t ⇒ cpu_id_t ⇒ mpss_t ⇒ bool
csb_next r c s ≡ ∃rl. r · rl = mpss_csb s c

csb_dequeue :: cpu_id_t ⇒ mpss_t ⇒ mpss_t
csb_dequeue c s ≡ s(|mpss_csb := (mpss_csb s)(c := tl (mpss_csb s c))|)

csb_enqueue :: p_region_t ⇒ cpu_id_t ⇒ mpss_t ⇒ mpss_t
csb_enqueue r c s ≡ s(|mpss_csb := (mpss_csb s)(c := mpss_csb s c @ [r])|)

csb_contains :: p_region_t ⇒ cpu_id_t ⇒ mpss_t ⇒ bool
csb_contains r c s ≡ r ⊆ ⋃set (mpss_csb s c)

csb_empty :: cpu_id_t ⇒ mpss_t ⇒ bool
csb_empty c s ≡ mpss_csb s c = []
```

With the predicate csb_next we can test if r is the atomic memory access that will next be drained from the store buffer of CPU c. The modifier csb_dequeue removes the next memory access from the store buffer of CPU c, whereas the modifier csb_enqueue enqueues the memory access r into that store buffer. Predicate csb_contains tests whether the store buffer contains the memory addresses in r, while predicate csb_empty tests for an empty store buffer.

Definition 5.2.7 (Execution of Abstract Instructions)

```
exec_read :: p_region_t ⇒ cpu_id_t ⇒ mpss_t ⇒ mpss_t
exec_read r c s ≡ s(|mpss_al := mpss_al s @ [AccRead r c]|)

exec_write :: p_region_t ⇒ cpu_id_t ⇒ mpss_t ⇒ mpss_t
exec_write r c s ≡ s(|mpss_al := mpss_al s @ [AccWrite r c]|)

exec_start_cpu :: cpu_id_t ⇒ instr_t list ⇒ mpss_t ⇒ mpss_t
exec_start_cpu cn il s ≡ if mpss_cr s cn then s
                         else s(|mpss_cr := (mpss_cr s)(cn := True),
                                 mpss_cil := (mpss_cil s)(cn := il)|)
```

Applying a read or write to the memory subsystem is modelled as appending it to the current memory-access scenario. Starting CPU cn is defined as setting the currently-running flag of that CPU to True and initialising its instruction list to il. If the CPU is already running, nothing is done.

With the necessary helper functions introduced, we define the transition relation trans_rel by means of the following six transition rules. Each rule applies for all system states s and for all CPUs c.

Definition 5.2.8 *Reading directly from the memory subsystem:*

$$\frac{\text{cil\_next}\ (\text{InstrRead}\ r)\ c\ s \qquad \neg\ \text{csb\_contains}\ r\ c\ s}{(s,\ \text{exec\_read}\ r\ c\ (\text{cil\_dequeue}\ c\ s)) \in \text{trans\_rel}}$$

If the next instruction to be executed is a read of memory addresses r, and r is not in the store buffer, we dequeue the instruction and read from the memory subsystem.

Definition 5.2.9 *Reading from the store buffer (store-buffer forwarding):*

$$\frac{\text{cil\_next}\ (\text{InstrRead}\ r)\ c\ s \qquad \text{csb\_contains}\ r\ c\ s}{(s,\ \text{cil\_dequeue}\ c\ s) \in \text{trans\_rel}}$$

If the next instruction to be executed is a read of memory addresses r, and r is in the store buffer, we dequeue the instruction but do not read from the memory subsystem.

Definition 5.2.10 *Writing to the store buffer:*

$$\frac{\text{cil\_next}\ (\text{InstrWrite}\ r)\ c\ s}{(s,\ \text{csb\_enqueue}\ r\ c\ (\text{cil\_dequeue}\ c\ s)) \in \text{trans\_rel}}$$

If the next instruction to be executed is a write of memory addresses r, we dequeue the instruction and enqueue the write into the store buffer.

Definition 5.2.11 *Draining a write from the store buffer:*

$$\frac{\text{csb\_next}\ r\ c\ s}{(s,\ \text{exec\_write}\ r\ c\ (\text{csb\_dequeue}\ c\ s)) \in \text{trans\_rel}}$$

On CPU c, if the next write in the store buffer is r, it is dequeued from the store buffer and applied to the memory subsystem. This models that a write can be dequeued from the store buffer at any time in case the store buffer is not empty. In our model, the store buffer is limited neither in its size nor in the time it holds a write.

Definition 5.2.12 *Executing a memory fence:*

$$\frac{\text{cil\_next}\ \text{InstrMFENCE}\ c\ s \qquad \text{csb\_empty}\ c\ s}{(s,\ \text{cil\_dequeue}\ c\ s) \in \text{trans\_rel}}$$

If the next instruction to be executed is an InstrMFENCE and the store buffer is empty, we dequeue the instruction from the instruction list. This rule ensures that after an InstrMFENCE is executed (i.e. dequeued), the store buffer is empty. As there is no other rule that dequeues an InstrMFENCE, executing an InstrMFENCE forces the store buffer to be drained to the memory subsystem.

Definition 5.2.13 *Starting a CPU:*

$$\frac{\text{cil\_next}\ (\text{InstrStartCPU}\ cn\ il)\ c\ s}{(s,\ \text{exec\_start\_cpu}\ cn\ il\ (\text{cil\_dequeue}\ c\ s)) \in \text{trans\_rel}}$$

If the next instruction to be executed is an InstrStartCPU $cn\ il$, we dequeue the instruction and start the new CPU cn with its initial instruction list il.

Note that at any given time (i.e. system state), any CPU c can perform any of the above transitions when its rule's assumptions are met.

Execution in the TSO model is defined as follows.

Definition 5.2.14 (Execution)

```
execute :: mpss_t ⇒ mpss_t ⇒ bool
execute s ≡ trans_rel* `` {s}
```

We return the set of all possible system states that can be generated by starting to execute the TSO system from the initial state s. Note that trans_rel* denotes the reflexive transitive closure of the transition relation. This models the system non-deterministically transitioning to any possible consecutive system state as defined by the transition rules.

The user of the TSO model is free to specify any initial system state. For convenience, we define a canonical initial system state, which should be applicable in most bootstrapping situations where the system powers on with only one CPU running.

Definition 5.2.15 (Canonical Initial System State)

```
init_mpss :: nat ⇒ instr_t list ⇒ mpss_t
init_mpss c il ≡ (|mpss_cr = (λ_. False)(c := True), mpss_cil = (λ_. [])(c := il),
                   mpss_csb = λ_. [], mpss_al = []|)
```

The system starts with an empty memory-access scenario mpss_al, empty store buffers mpss_csb, and one CPU c running the instruction list il.

5.2.3 Specifying Abstract Parallel Programs

There are multiple ways to specify the TSO model's input (the abstract parallel program). It could, for example, be hand-written as a direct abstract specification of the program under verification. However, such a specification would be very abstract because of the limited expressiveness of the TSO model due to its small instruction set. Most details that we might want to prove theorems about cannot be modelled directly in the TSO model. For example, the TSO model has no way of storing actual values in memory addresses.

As already published in previous work [vT10], we propose to derive the abstract parallel program from another specification that is much richer in detail and able to store values in variables. More precisely, we propose to specify the program with non-deterministic state monads (Section 2.3) where we are free in the level of detail and variables we want to model. From this specification, we extract the abstract parallel program which we feed into the TSO model.

The extraction works as follows: Assume that the monad state type is a record, we add a field that holds an $instr\_t\ list$ (i.e. an abstract parallel program). We instrument

the specification such that whenever we read/write variables (i.e. memory), we append an InstrRead/InstrWrite to that list; and we append an InstrMFENCE or InstrStartCPU to model executing their implementation-counterparts. The instrumentation thereby generates a specification of the abstract parallel program which is tightly coupled with the monadic specification. We will use this approach in seL4::CMK's bootstrapping specification presented in Section 5.4.

The instrumentation has to be done manually in case of seL4, because the monadic language used in its specifications is *shallowly embedded*. With such an embedding, it is impossible to directly reason about the specification code, i.e. the presence/absence/position of specific instructions (e.g. memory reads/writes). It is only possible to reason about the state that is modified by the specification. Hence, there is no way of automatically extracting an abstract parallel program. In contrast, *deeply embedded* specification languages allow reasoning about the specification code itself. For such a language, instrumentation can be automated. Note that L4.verified uses a deep embedding for the Isabelle-internal representation of the C implementation.

As we will see in Section 5.4, the effort of manually instrumenting seL4::CMK's abstract bootstrapping specification is small. The main reason is that all reads/writes of kernel object memory are done via a handful of dedicated accessor functions. Therefore, it is sufficient to instrument these functions.

The potential issue with instrumenting state monads as described above is the discrepancy between *big-step semantics* and *small-step semantics*. State monads, as defined in Section 2.3, have big-step semantics, which means that in the postcondition of a Hoare triple, we can only reason about the final state after the monadic execution, but not about intermediate states. The "big step" is the program between the pre- and postcondition of a Hoare triple. In order to be able to reason about concurrent memory access, the TSO model follows small-step semantics where we can reason about every intermediate (concurrent) state of the system.

Specifications based on the type of state monad we describe in Section 2.3 assume that during the monadic execution, we observe *sequential semantics*. This means that the values stored in variables do not change unless we change them. In other words: All reads observe the last write.

For large parts of typical parallel programs, this assumption holds. It is a common pattern in parallel programming to have large stretches of program execution only accessing variables local to the *thread of execution*.[5] Between those stretches, we have concentrated concurrent data access, for example for acquiring/releasing locks or modifying lock-free data structures. In seL4::CMK's bootstrapping phase, for example, the concurrent data access happens in part 1 when writing the shared configuration data and starting CPUs which then concurrently read the configuration. The large stretch of sequential execution is part 2 where each node is initialised by its own CPU.

As a consequence, in order to be able to make the assumption that sequential semantics are observed during the monadic execution—and thereby connect big- and small-step semantics—we need to prove that the abstract parallel program generated by the specification does not lead to memory-access scenarios that violate sequential semantics. To this end, we provide the *sequential-semantics theorem* presented in Section 5.3. As we will see in that section, the requirement for sequential semantics does not preclude concurrent execution and memory access.

[5]The definition of a *thread of execution* varies with the level of abstraction. On the lowest implementation level, it is a CPU. On top of a kernel, it is a proper thread. In a monadic abstract specification, it is the state monad.

5.2.4 Memory/IPI Reordering

Starting CPUs is commonly implemented in hardware with dedicated IPIs (inter-processor interrupts). It is therefore important to know how IPIs and memory writes are ordered on a particular architecture. For example, CPU A writes some configuration data to memory and starts CPU B, which in turn reads that configuration from memory. Is CPU B guaranteed to read the data that CPU A had written? Or is it possible that the IPI is reordered with the writes of the data?

Surprisingly, the AMD and Intel Developer Manuals [AMDb, Inta] do not specify whether such reordering can happen or not. However, the programming community seems to rely on it not happening. Furthermore, our question in the AMD Developer Central [AMDa] was answered by someone who seems to be an AMD employee and wrote that reordering of IPIs and memory writes cannot happen as long as normal cacheable memory is used.[6] The same question in the Intel Developer Zone [Intb] did not yield any useful replies.

Consequently, we chose the conservative approach for the TSO model and made it possible that memory writes stay in the store buffer while InstrStartCPU instructions are executed. We argue that as long as it is not explicitly specified by the manufacturer, it is unwise to include assumptions about memory ordering in a formal model. Furthermore, the TSO model should not be specific to x86/x64 but also be applicable to TSO architectures where this kind of reordering may be explicitly allowed.

The consequence of allowing IPI reordering in the model is that InstrMFENCE instructions have to be placed correctly in the specification (and the implementation); and their correct placement has to be proved in order to be able to assert absence of unwanted memory-access scenarios.

5.2.5 Advantages and Limitations

The most restrictive limitation is that the TSO model only stores memory-access scenarios, but not memory contents. We cannot use the TSO model to reason about parallel programs using arbitrary synchronisation mechanisms. It is not possible to model locking or lock-free data structures. The only synchronising abstract instruction is InstrStartCPU. We are restricted to reasoning about parallel programs that observe sequential semantics (such as the bootstrapping phase of seL4::CMK).

Nevertheless, reasoning about such programs is simplified because we decouple reasoning about program logic (variable values) from reasoning about concurrency (proving that sequential semantics are observed). Furthermore, reuse is facilitated by the original monad state not having to be changed except for adding the `instr_t list` field.

5.2.6 Discussion of Extensibility

The limitations presented in the section above are not inherent to the way we model TSO. Theoretically, it would be possible to enhance our TSO model to also store the contents of the memory. Combined with additional abstract instructions working with these contents, this would make it possible to reason about parallel programs using variables to synchronise concurrent actions. However, the drawback of using the TSO model to store program state is that when reading/writing abstract objects, their content

[6] x86/x64 CPUs allow specifying memory as *write combine* which results in a weaker memory model than TSO. This is not in the scope of our TSO model.

has to be mapped to/from byte values. This defeats one of the main purposes of abstract specifications: being able to abstract away from low-level representation of data.

We believe that a good compromise is to divide up the program state into abstract objects and synchronisation variables. This division should be possible for most parallel programs unless program logic is too intermingled with synchronisation (e.g. lock-free data structures). The synchronisation variables on the level of the TSO model can then be used to reason about concurrency, e.g. to prove deadlock freedom or that when accessing abstract objects in critical sections, sequential semantics are observed. It would also be possible to directly model architecture-provided synchronisation primitives such as CAS (compare-and-swap) or TAS (test-and-set).

The memory subsystem is currently modelled as a piece of sequentially consistent memory, which is sufficient to formulate theorems such as the sequential-semantics theorem. However, we could explicitly model the memory subsystem by adding caches and their coherence protocols to the model. This would enable reasoning about cache behaviour such as cache-line migration/bouncing/collisions and cache hits/misses.

5.2.7 Summary

In Section 5.2.2, we introduced our TSO multiprocessor execution model, which augments the existing verification framework with support for concurrency. This model non-deterministically and symbolically runs an abstract parallel program and computes all possible resulting memory-access scenarios.

The TSO model can be used to prove presence or absence of memory-access scenarios resulting from symbolically running a given abstract parallel program. In Section 5.2.3, we presented our approach to specifying such abstract parallel programs by extracting them from monadic specifications.

We elaborated on the issue of memory/IPI reordering and the resulting requirement of having to prove correct placement of InstrMFENCE instructions.

The multiprocessor execution model presented here is—to the best of our knowledge—the first to model CPUs starting other CPUs in presence of memory reordering.

5.3 Sequential-Semantics Theorem

We now present the kernel-agnostic *sequential-semantics theorem*, based on the TSO model introduced in Section 5.2.2. It is applicable in any bootstrapping situation where CPUs start other CPUs with each CPU bootstrapping its own isolated environment, such as clustered-multikernel bootstrapping.

The theorem states that if certain assumptions hold about the abstract parallel program fed into the TSO model, each CPU locally observes sequential semantics in every possible memory-access scenario. As we will see in this section, this does not preclude concurrent execution and memory access. It allows a restricted form of concurrency.

The aforementioned assumptions are: (1) memory fences are used correctly, and (2) a specific memory-access pattern is adhered to. These assumptions have to be proved for the abstract parallel program in question when applying this theorem. We will apply it in Section 5.5 to prove the seL4::CMK-specific *kernel-memory-sequential-access theorem*.

5.3.1 Definition

First, we have to define what it means for a CPU to observe sequential semantics. Informally, it means that from a CPU's point of view, memory content does not change without that CPU changing it itself. In other words: Locally, all reads observe the last write. Formally, the sequential-semantics property is defined as follows.

Definition 5.3.1 (Sequential-Semantics Property)

```
observes_seq_sem :: cpu_id_t ⇒ instr_t list ⇒ bool
observes_seq_sem c il ≡ ∀al∈mpss_al ` execute (init_mpss c il). al_has_seq_sem al
```

When starting the abstract parallel program `il` on the first CPU `c`, its execution only produces memory-access scenarios `al` which have sequential semantics.

Definition 5.3.2 (Sequential Semantics of a Memory-Access Scenario)

```
al_of_paddr :: paddr_t ⇒ acc_t list ⇒ acc_t list
al_of_paddr p al ≡ [a←al . p ∈ p_reg_of_acc a]

al_has_seq_sem :: acc_t list ⇒ bool
al_has_seq_sem al ≡
    ∀p c i j.
      let alp = al_of_paddr p al
      in i < j ∧
         j < length alp ∧
         cpu_of_acc alp[i] = c ∧ cpu_of_acc alp[j] = c ∧ is_acc_read alp[j] ⟶
         ¬ (∃k>i. k < j ∧ cpu_of_acc alp[k] ≠ c ∧ ¬ is_acc_read alp[k])
```

The memory-access scenario `al` has sequential semantics if, for all memory addresses `p` and CPUs `c`, between every read/write $alp_{[i]}$ followed by a later read $alp_{[j]}$ of that CPU, no write $alp_{[k]}$ by another CPU occurs.

Note that p_reg_of_acc returns the accessed memory region of an `acc_t`, which al_of_paddr uses to filter out memory accesses in `al` that do not involve memory address `p`. The function cpu_of_acc returns the ID of the CPU that performed the specific memory access. The predicate is_acc_read returns True if the memory access in question is an AccRead, and False if it is an AccWrite.

With the predicate observes_seq_sem we have now defined the goal of the sequential-semantics theorem, i.e. that each CPU locally observes sequential semantics in every possible memory-access scenario arising from executing a specific abstract parallel program. In the assumptions of the theorem, we want to specify properties about the program that are strong enough for the theorem to hold, but not too strong in order for the theorem to be widely applicable.

We choose a property that is specific to a bootstrapping situation where CPUs start other CPUs with each CPU bootstrapping its own isolated environment. It asserts that whenever the program of a CPU A instructs it to start another CPU B, the program given to CPU B will only write to memory addresses that the remaining program of CPU A will not read from, and vice versa. Note that it is therefore permissible for both programs to only read or only write from/to a specific memory address. The property is recursive in case any of the CPUs' remaining programs contain further InstrStartCPU instructions.

We call this the *disjoint-read/write-region property*. It resembles the definition of *data-race freedom* from Sewell et al. [SSO+10]: "[Memory operations are] *competing* if they access the same address, one is a write, and the other is a read ([...] it is not necessary to

consider write/write pairs as competing). We say that a program is data race free if it is impossible for a competing read/write pair to execute back-to-back." Both definitions have in common that competing read/read or write/write accesses do not cause data races, only competing read/write accesses do.

The disjoint-read/write-property seems very restrictive at first sight. However, it not only allows each CPU bootstrapping (initialising) its own dedicated region of memory, it also allows multiple CPUs to read concurrently from the same memory addresses. This makes sense, for example, if these memory addresses are written before the concurrently reading CPUs are started. This is the situation in seL4::CMK's part 1 of the bootstrapping phase, where the first CPU writes global configuration data before it starts the other CPUs that will later read this configuration concurrently. The property is recursive and therefore also applicable to the nested starting of CPUs that occurs in seL4::CMK: Each node's first CPU starts the remaining CPUs of the node after the node's kernel data is initialised.

Definition 5.3.3 (Disjoint-Read/Write-Region Property)

```
instr_rw_p_regs_disj :: instr_t list ⇒ bool
instr_rw_p_regs_disj [] = True
instr_rw_p_regs_disj (InstrRead _ · il) = instr_rw_p_regs_disj il
instr_rw_p_regs_disj (InstrWrite _ · il) = instr_rw_p_regs_disj il
instr_rw_p_regs_disj (InstrMFENCE · il) = instr_rw_p_regs_disj il
instr_rw_p_regs_disj (InstrStartCPU _ start_il · il) =
    instr_read_p_reg il ∩ instr_write_p_reg start_il = ∅ ∧
    instr_write_p_reg il ∩ instr_read_p_reg start_il = ∅ ∧
    instr_rw_p_regs_disj start_il ∧ instr_rw_p_regs_disj il
```

The property is defined recursively with each recursion handling the first instruction of the abstract parallel program. The instructions InstrRead, InstrWrite and InstrMFENCE have no direct impact on the property here. The interesting case is the InstrStartCPU instruction. We have to ensure that the *read/write regions* (instr_read_p_reg/instr_write_p_reg defined below) of the remaining program to be executed (`il`) and the new CPU's program (`start_il`) are disjoint. Furthermore, we assert that the property holds (recursively) for both of these programs.

The *read region* and *write region* of an abstract parallel program are defined as follows.

Definition 5.3.4 (Read Region)

```
instr_read_p_reg :: instr_t list ⇒ p_region_t
instr_read_p_reg [] = ∅
instr_read_p_reg (InstrRead p_reg · il) = instr_read_p_reg il ∪ p_reg
instr_read_p_reg (InstrWrite p_reg · il) = instr_read_p_reg il
instr_read_p_reg (InstrMFENCE · il) = instr_read_p_reg il
instr_read_p_reg (InstrStartCPU _ start_il · il) =
    instr_read_p_reg il ∪ instr_read_p_reg start_il
```

This function returns all memory addresses that are part of any InstrRead instruction in the program in question. Programs of new CPUs to be started are included recursively. The write region (function instr_write_p_reg) is defined accordingly.

In a sequentially consistent system (e.g. without store buffers), the disjoint-read/write-region property would be a sufficient assumption for the sequential-semantics theorem. In TSO, however, we require correct placement of memory fences in order for the theorem to be true. For example, a CPU A might write to a memory address and start CPU B, which reads from that address multiple times. CPU A's write might be delayed in the store buffer

and be written to the memory subsystem between two reads of CPU B. This means that CPU B does not observe sequential semantics anymore, even though the program satisfies the disjoint-read/write-region property.

Consequently, we need a second assumption for the sequential-semantics theorem, which ensures correct placement of memory fences. The rule for correct placement is simple: Each InstrStartCPU needs to be proceeded by an InstrMFENCE.

Definition 5.3.5 (Correct Memory-Fence Placement)

```
instr_mfence_start_cpu :: instr_t list ⇒ bool
instr_mfence_start_cpu [] = True
instr_mfence_start_cpu (InstrMFENCE · InstrStartCPU _ start_il · il) =
      instr_mfence_start_cpu il ∧ instr_mfence_start_cpu start_il
instr_mfence_start_cpu (InstrStartCPU _ _ · il) = False
instr_mfence_start_cpu (_ · il) = instr_mfence_start_cpu il
```

An InstrStartCPU is only allowed to occur if it is preceded by an InstrMFENCE. The property needs to hold recursively for all programs started on all new CPUs. Note that Isabelle tries to match the equations top to bottom. The last one is a catch-all rule.

With the two assumptions and the goal defined, we are now ready to state the sequential-semantics theorem.

Theorem 5.3.6 (Sequential Semantics)

$$\frac{\texttt{instr\_mfence\_start\_cpu}\ il \qquad \texttt{instr\_rw\_p\_regs\_disj}\ il}{\texttt{observes\_seq\_sem}\ c\ il}$$

If the disjoint-read/write-region property holds and memory fences are correctly placed in an abstract parallel program *il*, executing it can only result in memory-access scenarios in which each CPU locally observes sequential semantics.

Showing the proof of this theorem and discussing its complexity is deferred to the end of this chapter (Section 5.12.1).

5.3.2 Summary

We contributed the kernel-agnostic sequential-semantics theorem. The theorem uses the TSO model to state that if certain assumptions hold about the abstract parallel program fed into the TSO model, each CPU locally observes sequential semantics in every possible memory-access scenario. The assumptions are: (1) memory fences are used correctly, and (2) the disjoint-read/write-region property is adhered to. These assumptions have to be proved for the abstract parallel program in question when applying this theorem. We will apply it in Section 5.5 to prove the seL4::CMK-specific *kernel-memory-sequential-access theorem*.

5.4 Bootstrapping Specification

In this section, we present (1) our approach to formally connecting a bootstrapping specification to the system boot loader, (2) a model of how such a specification dynamically allocates and accesses data structures, (3) how such a specification is instrumented[7], and based on the above, (4) an abstract specification of bootstrapping a clustered multikernel, with the example of seL4::CMK.

[7] We explained the goal of this instrumentation in Section 5.2.3.

Recall from Section 4.5 that part 1 of seL4::CMK's bootstrapping phase is responsible for discovering available resources (CPUs, memory, devices), initialising the platform, writing down the system configuration (CPU/memory assignment to nodes) and starting the remaining CPUs. Each node's first CPU reads the system configuration and then calls part 2 of bootstrapping, passing the node's configuration as function-call arguments. Part 2 is almost identical to the uniprocessor bootstrapping and is not exposed to any concurrency because all data accesses are node-local.

Bootstrapping was not in the scope of the L4.verified project (Section 2.6.1), which means that we are not able to leverage an existing bootstrapping specification as intended by the refinement lifting framework. Therefore, we wrote the abstract specification of both bootstrapping parts from scratch. In contrast, applying the refinement lifting framework to a uniprocessor kernel with an existing bootstrapping specification would only require part 1 to be written from scratch. For part 2, we would be able to use the existing uniprocessor bootstrapping specification with only minor modifications.

5.4.1 Bootstrapping Part 1

When a multiprocessor system is powered up, the initial CPU is started first. This CPU then runs a boot loader which fetches the *kernel image* from ROM, disk or network and loads it into memory. Some boot loaders support *boot modules*, which are additionally loaded into memory. After that, the kernel's bootstrapping code is called. In case of seL4::CMK, the boot loader has to provide one boot module per node, each containing the *user-level image* of that particular node. A user-level image contains all static code and data necessary to run a node's initial thread.

The boot loader also has to provide platform configuration data to the kernel, which includes: (1) the IDs of the available (but not yet started) CPUs, (2) the available physical memory, (3) the region of physical memory where the kernel image was loaded to, and (4) where the boot modules were loaded to.

In the abstract specification, we model the boot loader's written configuration data as global constants.

Definition 5.4.1 (Configuration Constants)

```
types vptr_t = nat

record ui_info_t =  ui_info_p_reg      :: p_region_t
                    ui_info_pv_offset  :: nat
                    ui_info_v_entry    :: vptr_t

plat_avail_p_reg  :: p_region_t
ki_p_reg          :: p_region_t
ui_info_list      :: ui_info_t list

num_nodes     :: nat
num_cpu_list  :: nat list
cpu_list      :: cpu_id_t list list
```

The available physical memory is stored in plat_avail_p_reg. The region where the kernel image was loaded to is ki_p_reg. The list ui_info_list stores one `ui_info_t` record per node. That record's field ui_info_p_reg stores the memory region where the user-level image was loaded. The kernel also needs to know at which virtual address to map the user-level image (ui_info_pv_offset) and the initial thread's entry pointer (ui_info_v_entry).

The configured number of nodes is stored in num_nodes. For each node, the number of assigned CPUs is stored in the list num_cpu_list. Finally, the nested list cpu_list defines which CPUs are assigned to which nodes.

In order to be able to reason about these constants or about the program state derived from them, we need to know their (possible) values. However, simply assigning specific values in the constants' definitions is not desirable because it would restrict the proof to be valid only for that particular system configuration. However, we want the proof to be applicable for all possible system configurations. To this end, we *underspecify* the constants, i.e. instead of defining them, we only declare them and add a set of axioms about them. Therefore, these axioms model the behaviour of the boot loader. The axioms (presented below) are carefully chosen: (1) We only introduce axioms that are actually needed by the proof. (2) The axioms only talk about these constants. (3) The axioms should be as simple as possible.

Axiom 5.4.2 *The number of nodes is greater than zero.*

$$0 < \text{num\_nodes}$$

Axiom 5.4.3 *The list* num_cpu_list *has correct length, and each node is assigned at least one CPU.*

$$\text{length num\_cpu\_list} = \text{num\_nodes} \wedge (\forall n{<}\text{num\_nodes}.\ 0 < \text{num\_cpu\_list}_{[n]})$$

Axiom 5.4.4 *The (nested) lists in* cpu_list *have correct lengths, and each CPU is assigned to only one node.*

$$\begin{aligned}&\text{length cpu\_list} = \text{num\_nodes} \wedge\\ &(\forall n{<}\text{num\_nodes}.\ \text{length cpu\_list}_{[n]} = \text{num\_cpu\_list}_{[n]}) \wedge\\ &(\forall n1{<}\text{num\_nodes}.\\ &\quad \forall n2{<}\text{num\_nodes}.\ n1 \neq n2 \longrightarrow \text{set cpu\_list}_{[n1]} \cap \text{set cpu\_list}_{[n2]} = \emptyset)\end{aligned}$$

Axiom 5.4.5 *The available memory is finite.*

$$\text{finite plat\_avail\_p\_reg}$$

Axiom 5.4.6 *The list* ui_info_list *has correct length.*

$$\text{length ui\_info\_list} = \text{num\_nodes}$$

Axiom 5.4.7 *All user-level images are located in the available memory.*

$$\forall node\_id{<}\text{num\_nodes}.\ \text{ui\_info\_p\_reg ui\_info\_list}_{[node\_id]} \subseteq \text{plat\_avail\_p\_reg}$$

Axiom 5.4.8 *Each user-level image has a finite size.*

$$\forall node\_id{<}\text{num\_nodes}.\ \text{finite } (\text{ui\_info\_p\_reg ui\_info\_list}_{[node\_id]})$$

Axiom 5.4.9 *No user-level image overlaps with the kernel image.*

$$\forall node\_id{<}\text{num\_nodes}.\ \text{ui\_info\_p\_reg ui\_info\_list}_{[node\_id]} \cap \text{ki\_p\_reg} = \emptyset$$

Axiom 5.4.10 *The user-level images do not overlap with each other.*

$$\begin{aligned}&\forall node\_1\_id{<}\text{num\_nodes}.\\ &\quad \forall node\_2\_id{<}\text{num\_nodes}.\\ &\qquad node\_1\_id \neq node\_2\_id \longrightarrow\\ &\qquad \text{ui\_info\_p\_reg ui\_info\_list}_{[node\_1\_id]} \cap \text{ui\_info\_p\_reg ui\_info\_list}_{[node\_2\_id]} = \emptyset\end{aligned}$$

Monad State

With the initial state defined (or underspecified), we are now ready to specify the kernel's bootstrapping. We use the non-deterministic state monad from Section 2.3 and define the following monad state and type.

Definition 5.4.11 (Global Monad State Type)

```
types pmem_var_t = paddr_t ⇀ var_t

record glms_t =  glms_pmem_var  :: pmem_var_t
                 glms_cpu_id    :: cpu_id_t
                 glms_il        :: instr_t list

types 'r glms_monad_t = (glms_t, 'r) nondet_monad
```

The first field of the monad state record models global memory content on the level of typed abstract variables, i.e. above byte-encoded representation. It is a partial function from physical memory addresses to variables. The function returns None of no value is stored, otherwise it returns a variable of type `var_t` (defined below). The field glms_cpu_id returns the ID of the CPU currently executing the specified code. Reading this field models querying the CPU for its ID (with architecture-specific assembly instructions). The third field holds the abstract parallel program that is created by the instrumented monadic specification (Section 5.2.3).

We define the following initial global monad state.

Definition 5.4.12 (Initial Global Monad State)

```
init_glms :: glms_t
init_glms ≡ (|glms_pmem_var = empty, glms_cpu_id = cpu_list[0][0], glms_il = []|)
```

This models the system powering on with empty memory content. The CPU running initially is the one assigned to be the first CPU of the first node (the CPU's ID itself is still underspecified). The assignment is done by the boot loader. The abstract parallel program glms_il is empty and ready to be accumulated by the instrumentation.

Definition 5.4.13 (Global-Variable Type)

```
datatype var_t =  VarGlKs glks_t
                | VarNdKs ndks_t
                | VarKHeap
```

We define three *top-level variable types*: The first type holds the system configuration (aka static **gl**obal **k**ernel **s**tate, `glks_t`) and is used by part 1 of bootstrapping. The second type (`ndks_t`) holds a **n**o**d**e's static **k**ernel **s**tate. The VarKHeap type does not carry any content directly. If a memory address contains a VarKHeap value, it means that this memory address is occupied by a kernel object in the kernel heap. The contents of `ndks_t` and the kernel heap are specific to part 2 of bootstrapping and are therefore discussed in Section 5.4.2. The reason for additionally storing a VarKHeap value at this level is to model wild writes. These are unintentional writes to wrong memory addresses, which can overwrite important data. Consequently, Isabelle will require us to prove that no such wild writes happen to memory addresses that contain data which the property we are proving depends on.

In the C implementation, `glks_t` and `ndks_t` are structs that are statically allocated by the compiler. We model their implicit C pointers by global Isabelle constants of type `paddr_t`.

The singleton allocation of struct `glks_t` is modelled as follows.

Definition 5.4.14

```
var_glks :: paddr_t
glks_size :: nat

glks_p_reg :: p_region_t
glks_p_reg ≡ {var_glks..<var_glks + glks_size}
```

The allocations of struct `ndks_t` are modelled as follows.

Definition 5.4.15

```
var_ndks_list :: paddr_t list
ndks_size :: nat

ndks_p_reg :: node_id_t ⇒ p_region_t
ndks_p_reg node_id ≡ {var_ndks_list[node_id]..<var_ndks_list[node_id] + ndks_size}
```

The list var_ndks_list contains a pointer to each node's static kernel state, i.e. each node's allocated `ndks_t`. The type `node_id_t` is defined as `nat`.

Just as the configuration data provided by the boot loader, the compiler-provided pointers and sizes are underspecified as well. The axioms are defined in the same way.

Axiom 5.4.16 *The sizes of `glks_t` and `ndks_t` are greater than zero.*

```
0 < glks_size
0 < ndks_size
```

Axiom 5.4.17 *The static global kernel state is located within the kernel image.*

```
glks_p_reg ⊆ ki_p_reg
```

Axiom 5.4.18 *Each node's static kernel state is located within the kernel image.*

```
∀ node_id<num_nodes. ndks_p_reg node_id ⊆ ki_p_reg
```

Axiom 5.4.19 *The nodes' static kernel states do not overlap with each other.*

```
∀ node_1_id<num_nodes.
   ∀ node_2_id<num_nodes.
      node_1_id ≠ node_2_id ⟶
      ndks_p_reg node_1_id ∩ ndks_p_reg node_2_id = ∅
```

Axiom 5.4.20 *None of the nodes' static kernel states overlap with the static global kernel state.*

```
∀ node_id<num_nodes. glks_p_reg ∩ ndks_p_reg node_id = ∅
```

In order to facilitate reading and writing of `glks_t` and `ndks_t`, we define four monadic accessor functions.

Reading/writing a `glks_t` variable from/to physical address `paddr` is defined as follows.

Definition 5.4.21

```
var_read_glks :: paddr_t ⇒ glks_t glms_monad_t
var_read_glks paddr ≡
    do pmem_var ← gets glms_pmem_var;
    case pmem_var paddr of None ⇒ select UNIV
        | ⌊VarGlKs val⌋ ⇒
            do append_instr_read_reg {paddr..<paddr + glks_size};
               return val
            od
        | ⌊_⌋ ⇒ select UNIV
    od

var_write_glks :: paddr_t ⇒ glks_t ⇒ unit glms_monad_t
var_write_glks paddr val ≡
    do modify (λm. m(|glms_pmem_var := glms_pmem_var m(paddr ↦ VarGlKs val)|));
    append_instr_write_reg {paddr..<paddr + glks_size}
    od
```

Reading/writing an `ndks_t` variable from/to physical address `paddr` is defined as follows.

Definition 5.4.22

```
var_read_ndks :: paddr_t ⇒ ndks_t glms_monad_t
var_read_ndks paddr ≡
    do pmem_var ← gets glms_pmem_var;
    case pmem_var paddr of None ⇒ select UNIV
        | ⌊VarNdKs val⌋ ⇒
            do append_instr_read_reg {paddr..<paddr + ndks_size};
               return val
            od
        | ⌊_⌋ ⇒ select UNIV
    od

var_write_ndks :: paddr_t ⇒ ndks_t ⇒ unit glms_monad_t
var_write_ndks paddr val ≡
    do modify (λm. m(|glms_pmem_var := glms_pmem_var m(paddr ↦ VarNdKs val)|));
    append_instr_write_reg {paddr..<paddr + ndks_size}
    od
```

Note that the value of a variable is only stored in the first memory address byte, even though the (underspecified) size of the variable might be bigger than 1 byte. It reflects the reality in the sense that when using C pointers, we also point at the first memory address byte. On the other hand, it does not model accurately what happens when someone accidentally directly modifies the non-first memory address occupied by the variable. However, this limitation can be alleviated. For example, seL4 stores kernel objects in the same way and invariants ensure that such accidental writes cannot happen.

When reading from a memory address that does contain the expected variable type or has no value stored at all, select UNIV returns the non-deterministic set of all possible values. This models that reading from such a memory address might return any value.

Instrumentation

Besides reading/writing the actual values, the accessor functions presented above call the functions append_instr_read_reg and append_instr_write_reg. These calls are the instrumentation to generate the abstract parallel program mentioned earlier.

Appending InstrRead and InstrWrite to the abstract parallel program is defined as follows.

Definition 5.4.23

```
append_instr_read_reg :: p_region_t ⇒ unit glms_monad_t
append_instr_read_reg p_reg ≡
    do il ← select {il | ∀instr∈set il. ∃r. instr = InstrRead r ∧ r ⊆ p_reg};
    modify (λs. s(|glms_il := glms_il s @ il|))
    od
```

We select the non-deterministic set of all possible InstrRead sequences that read from memory region `p_reg` and append it to the already accumulated abstract parallel program. We define append_instr_write_reg in exactly the same way.

Overapproximation

With this way of instrumenting, we *overapproximate* memory accesses in the abstract specification. This means that for abstract read/write operations, we do not specify in which order the variable's memory addresses are read/written by the final assembly code. This applies between reads and between writes. The order of reads vs. writes is preserved.

We have to use overapproximation for two reasons: (1) Different ways of implementing these operations in C result in different access patterns and (2) different C compilers emit different assembly code which also results in varying patterns.

We make the assumption that for the read/write operation, the C implementation does not generate reads/writes outside the variable's assigned memory region. Section 5.11.2 will discuss how this assumption can be removed.

Overapproximation is safe, under the assumption presented above. However, using it too extensively prevents a successful proof in case we do not know enough facts about the ordering of abstract instructions in order to prove a particular theorem. As we will see in Section 5.5, the degree of overapproximation we use allows proving the *kernel-memory-sequential-access theorem*. Note that it would not have been possible to prove the theorem if we had mixed reads and writes in our overapproximation.

System Bootstrapping Code

So far, we have presented the fundaments of the abstract specification: the memory model, the instrumentation to generate the abstract parallel program, and how we model compiler and boot loader. We now elaborate on the seL4::CMK-specific part of the abstract specification which directly corresponds to the C code, i.e. for each C function, we have an abstract counterpart with the same name. First, we introduce the central data structure of part 1 of bootstrapping. It contains the global configuration, which is named *shared state* in Figure 4.5.

Definition 5.4.24 (Global Kernel State)

```
types dev_p_regs_t = p_region_t ⇒ bool

record glms_t =  glks_avail_p_reg   :: p_region_t
                 glks_ki_p_reg      :: p_region_t
                 glks_sh_p_reg      :: p_region_t
                 glks_num_nodes     :: nat
                 glks_num_cpu_list  :: nat list
                 glks_cpu_list      :: cpu_id_t list list
                 glks_ui_info_list  :: ui_info_t list
                 glks_dev_p_regs    :: dev_p_regs_t
```

The data structures are a 1:1 match with the data structures on C level. Isabelle records represent C structs, a *nat* represents an `unsigned int` and an Isabelle list represents a C array. The only exception is *p_region_t*. In Isabelle, it is a set of physical addresses whereas in C, it is a struct with a `start` and `end` field.

Instead of defining the *glks_t* and *ndks_t* records, it would have been possible to define each of their fields as a proper top-level variable (*var_t*). This would have been necessary if the order in which they are accessed mattered proof-wise. However, combining the fields into records reduces proof complexity considerably. Doing this is permissible because we overapproximate memory accesses.

The top-level bootstrapping function is called boot_sys (type *unit glms_monad_t*). When it starts, in initialises the *glms_t* fields with the respective constants that model the boot loader's output. For example, the value of the constant cpu_list is assigned to the field glks_cpu_list, etc. This models the bootstrapping code extracting the configuration from the boot loaders data structures and writing them into seL4::CMK's own data structures. After that, boot_sys removes the kernel- and user-level-image regions from the available memory region glks_avail_p_reg in order for the latter to represent the *available memory* that can be assigned to the nodes. The configurable region of *shared memory*[8] is removed from glks_avail_p_reg as well and stored in glks_sh_p_reg. Next, memory-mapped devices are discovered and their memory regions written into glks_dev_p_regs. Finally, boot_sys starts each node's first CPU by calling start_cpu. Refer to Figure 4.5 for a graphical presentation of the function calls involved in bootstrapping.

Starting CPUs

The function start_cpu is a central point in bootstrapping, e.g. it contains the instrumentation that turns the abstract program into an abstract *parallel* program.

Definition 5.4.25 (CPU Start)

```
start_cpu :: cpu_id_t ⇒ unit glms_monad_t ⇒ unit glms_monad_t
start_cpu cpu_id start_fun ≡
    do append_instr InstrMFENCE;
       pmem_var ← gets glms_pmem_var;
       (ret', glms') ←
       select_f
         (start_fun
           (|glms_pmem_var = pmem_var, glms_cpu_id = cpu_id, glms_il = []|));
       modify (glms_pmem_var_update (λ_. glms_pmem_var glms'));
    append_instr (InstrStartCPU cpu_id (glms_il glms'))
    od
```

The first parameter is the ID of the CPU to be started. The second parameter is the monadic function that should be executed by the new CPU.

The first thing start_cpu does is executing a memory fence: It calls the function append_instr and instructs it to append an InstrMFENCE to the abstract parallel program.

Modelling a new CPU being started and executing *start_fun* is done by applying it to a crafted initial state. This state consists of the current memory state (*pmem_var*), the ID of the new CPU (*cpu_id*) and a yet empty abstract parallel program, just as in init_mpss.

The function select_f is defined as *λS s. (*fst *S* × *{s}*, snd *S)*. It lifts the resulting monad state of function *S* out of the monadic computation and returns it as proper return value without changing the current monad state *s*. With the help of select_f, we can extract

[8] This is the *shared memory* mentioned in Section 4.2, which will end up as user-level memory that is shared between nodes.

glms', the final monad state of *start_fun*. From this state, we extract the memory state glms_pmem_var and set it the new memory state. Finally, start_cpu appends an InstrStartCPU instruction to the abstract parallel program. This is where the program becomes parallel: Besides the ID of the started CPU, the InstrStartCPU also contains the abstract parallel program glms_il that was accumulated by *start_fun*.

In C, start_cpu is implemented by two calls to assembly-implemented functions. The first executes a memory fence, the second starts the new CPU identified by its ID *cpu_id* and instructs it to start executing at function pointer *start_fun*.

Node Bootstrapping Code

At its end, the main function boot_sys boots the first node by calling boot_node and calls start_cpu for each further node that has to be booted. The *start_fun* parameter of start_cpu is set to boot_node. Like this, every node's first CPU is now executing boot_node. The nodes' remaining CPUs are not started yet.

The boot_node function cannot have any parameters or return value because it is the first function executed by the started CPUs. Hence, the first thing it has to do is to confirm its identity by querying the CPU for its ID. From the global kernel state, it then finds its assigned *node_id* and extracts all necessary node-configuration information. Here, it also calls split_region, which splits up the total available memory between the nodes and returns the current node's available memory region.

The split_region function splits up a region of memory into *n* pieces of the same size. Its definition in the abstract specification is an exact line-by-line match with the C implementation. This increases the chances that a bug in this vital function is already caught on the abstract level by the isolation theorems presented later.

Finally, we transition into part 2 of bootstrapping by having boot_node call init_node_state, passing all node-configuration data as regular function-call arguments. The function init_node_state constitutes part 2 of bootstrapping and initialises the node (see next section). When it returns, boot_node uses start_cpu to start the remaining CPUs of its node.

5.4.2 Bootstrapping Part 2

Part 2 of bootstrapping is implemented in the function init_node_state. This function runs in a non-deterministic state monad with *ndks_t* being its monad state type. The monad type *ndks_monad_t* is defined accordingly. We directly use this state type because we only modify node-local data. While this simplifies specification of and proofs about init_node_state, it also means calling it from boot_node requires a conversion between monad state types. More specifically, the *ndks_monad_t* type needs to be *lifted* into the *glks_monad_t* type, which works as follows: We read the current node's *ndks_t* state from memory using var_read_ndks and use it as initial monad state when calling init_node_state. Upon return, we store the final *ndks_t* state back into memory using var_write_ndks.

So far, we have covered init_node_state being able to initialise its node's static kernel data (*ndks_t*), after having received the node's configuration via its parameters. However, seL4 requires certain kernel objects being created and initialised during bootstrapping. The exact allocations depend on the configuration, which in turn depends on the hardware (e.g. available memory). Therefore, init_node_state needs to perform *dynamic object allocation*. This is where the complexity of bootstrapping part 2 lies.

Kernel Heap

Kernel objects are stored in the *kernel heap*. Each node has its own kernel heap, which lives in the node's assigned region of available memory. The kernel heap is modelled as a partial function from memory address to kernel object. It is stored in the field ndks_kheap of record `ndks_t`).

For reading/writing kernel objects, we define the accessor functions read_obj and write_obj. These functions are instrumented to store all read/written (accessed) kernel heap addresses in the ghost[9] field ndks_kheap_acc. When init_node_state returns, we call append_instr_rw_reg, which is a combination of append_instr_read_reg and append_instr_write_reg (Definition 5.4.23): It does a select on the non-deterministic set of all possible sequences of InstrRead/InstrWrite that read/write addresses stored in ndks_kheap_acc.

This way, we overapproximate accessing kernel objects within a node such that we cannot reason anymore about the order in which they were read/written. However, this is not a problem since they are not accessed concurrently. Moreover, we can reduce proof complexity considerably.

For kernels with existing bootstrapping specification, minimally invasive instrumentation of kernel heap access also has the advantage of being able to adapt an existing uniprocessor bootstrapping specification with only minor modifications.

5.4.3 Discussion

Table 5.1 lists the size of seL4::CMK's abstract bootstrapping specification, divided up into three categories: (1) *Semantics* comprises the memory model, accessor functions and instrumentation; (2) *Program Types* are specifications of types present in C; and (3) *Program Code* means direct specifications of C functions.

Abstract Specification	Semantics	Program Types	Program Code	Total
Bootstrapping Part 1	200 LOC	10 LOC	200 LOC	410 LOC
Bootstrapping Part 2	70 LOC	280 LOC	520 LOC	870 LOC
Total	270 LOC	290 LOC	720 LOC	1280 LOC

Table 5.1: *Size of seL4::CMK's Abstract Bootstrapping Specification*

Table 5.2 lists the size of the bootstrapping C code.

C Implementation	Program Code
Bootstrapping Part 1	430 LOC
Bootstrapping Part 2	580 LOC
Total	1010 LOC

Table 5.2: *Size of seL4::CMK's Bootstrapping C Code (without type definitions and function declarations)*

Comparing the *Program Code* columns of both tables, we can see that the specification size is about 70 percent of the C code's size. For an abstract specification, this percentage is comparatively high. It can be explained with that fact that we model all object sizes and types and every occasion an object is dynamically allocated during bootstrapping. In large parts, the abstract specification is close to a line-by-line match with the C code. The

[9] Refer to Section 3.3.1 for the definition of *ghost* state.

30 percent that are abstracted away are mostly initialising of object content that does not influence bootstrapping control flow or memory use.

When implementing seL4::CMK's bootstrapping, we first wrote the entire abstract specification and then manually translated it to C. During the translation process, we filled in the details that were abstracted away. In our experience, doing it this way had several advantages: First, Isabelle's static type checking was able to detect most of the countless minor specification bugs very early. A considerable number of these bugs would not have been detected by a C compiler if we had started implementing in C. Second, translating the abstract specification into C code was straight-forward. Hence, the translation process was done very quickly.

However, filling in the abstracted-away details turned out to be more complex than expected. For example, we realised in a few occasions that due to certain object content being abstracted away in the specification, we missed dependencies that need to be observed during bootstrapping. Only when implementing these object initialisations in C, we realised that the order in which some objects were initialised in the abstract specification was wrong.

In summary, we experienced that the benefits and drawbacks of starting the implementation on the abstract level instead of C more or less outweigh each other. Hence, we believe a better approach would have been to start with an intermediate executable specification, which is exactly how seL4 was initially prototyped [EKD+07].

5.4.4 Summary

In Section 5.4, we presented (1) our approach to formally connect a bootstrapping specification to the system boot loader, (2) a model of how such a specification dynamically allocates and accesses data structures, (3) how such a specification is instrumented, and based on the above, (4) an abstract specification of bootstrapping a clustered multikernel, with the example of seL4::CMK.

5.5 Kernel-Memory-Sequential-Access Theorem

Recall from Section 5.2 that in order to be able to make the assumption that sequential semantics are observed during a monadic execution, we need to prove that the abstract parallel program generated by the abstract specification does not lead to memory-access scenarios that violate sequential semantics.

As a first step towards this goal, we proved the kernel-agnostic sequential-semantics theorem (Theorem 5.3.6). This theorem allows us prove that sequential semantics are observed (predicate observes_seq_sem, Definition 5.3.1) if we prove that the abstract parallel program extracted from the abstract specification satisfies the following two assumptions: (1) memory fences are placed correctly (predicate instr_mfence_start_cpu, Definition 5.3.5) and (2) the disjoint-read/write-region property (predicate instr_rw_p_regs_disj, Definition 5.3.3) is adhered to.

We can now apply the sequential-semantics theorem and prove what we call the *kernel-memory-sequential-access theorem*, which states that seL4::CMK's bootstrapping is correct with regards to concurrency, i.e. that when accessing kernel memory during bootstrapping, we always observe sequential semantics.

5.5.1 Definition

Theorem 5.5.1 (Kernel Memory Sequential Access)

$$\{\!|\lambda s.\ s = \mathsf{init\_glms}|\!\}\ \mathsf{boot\_sys}\ \{\!|\lambda\_\ s.\ \mathsf{observes\_seq\_sem}\ \mathsf{cpu\_list}_{[0][0]}\ (\mathsf{glms\_il}\ s)|\!\}$$

In the precondition, we initialise the system as described in the definition of init_glms (Definition 5.4.12). For the postcondition, recall from Section 5.4 that boot_sys is the main bootstrapping function. Its returned monad state contains the entire initialised system state, and therefore, glms_il s returns the abstract parallel program of the entire bootstrapping. The first parameter of observes_seq_sem is the ID of the CPU where the program is started, which is the first CPU assigned to the first node (as defined in init_glms).

Showing and discussing the proof of the aforementioned assumptions—and therefore this theorem—is deferred to the end of this chapter (Section 5.12.2).

5.5.2 Summary

The kernel-memory-sequential-access theorem states that the abstract parallel program generated by seL4::CMK's boostrapping specification does not lead to memory-access scenarios that violate sequential semantics. This was the assumption we made by using a non-deterministic state monad for seL4::CMK's abstract specification. We have now proved that this assumption is correct.

5.6 Intermediate Conclusion

The chapter so far We have introduced our TSO *multiprocessor execution model*, which augments the existing verification framework with support for concurrency. This model non-deterministically and symbolically runs an abstract parallel program and computes all possible resulting memory-access scenarios.

The state monads used in the specifications of seL4 and seL4::CMK assume that during the monadic execution, we observe sequential semantics. As a first step towards proving this assumption, we contributed the kernel-agnostic *sequential-semantics theorem* (Section 5.3), which uses the TSO model to prove that in a bootstrapping situation, every CPU observes sequential semantics, under the assumption that (1) memory fences are used correctly and (2) the disjoint-read/write-region property is adhered to. These assumptions have to be proved for the specification in question when applying this theorem.

Section 5.4 was devoted to formally specifying the code of a clustered multikernel's bootstrapping phase. First, we presented our approach to connecting a bootstrapping specification to the system boot loader. Second, we showed how a specification dynamically allocates and accesses data structures. Third, we explained how we connect a specification to the multiprocessor execution model. We did all of this with the example of an abstract specification of seL4::CMK's bootstrapping, which we provided as well.

In Section 5.5, we combined this specification with the sequential-semantics theorem. The result was the *kernel-memory-sequential-access theorem*, which states that during bootstrapping of seL4::CMK, each CPU observes sequential semantics. This was the assumption we made by using a non-deterministic state monad for seL4::CMK's abstract specification. We therefore proved that this assumption is correct.

Conclusion So far, this chapter focused on correctness of the bootstrapping phase itself. The remainder of this chapter will talk about concepts, techniques and theorems that—while still talking about bootstrapping—are required by the runtime phase of the kernel.

Remainder of this chapter Section 5.7 introduces the *kernel isolation theorem*, which states that the nodes' kernels are bootstrapped isolated.

In Section 5.8, we present the *user-level isolation theorem* and the *user-level sharing theorem*. They assert useful properties about user-level memory, which a user-level application can rely on. For example, together with the kernel isolation theorem, the user-level isolation theorem provides node-isolation guarantees which can be useful for building systems on top of seL4::CMK. Nevertheless, the user-level theorems are not strictly necessary for refinement lifting.

In Section 5.9 we introduce a technique to connect seL4::CMK's abstract bootstrapping specification with the runtime-phase specification of seL4 (from the L4.verified project). This is necessary because the two specifications are based on slightly different formalisations.

Finally, in Section 5.10, we present the *multikernel refinement theorem*. This theorem shows refinement of a multikernel-version of seL4, i.e. a configuration of seL4::CMK where each node runs on exactly one CPU. The step to a *clustered multikernel*—i.e. multiple CPUs per node—is the topic of the next chapter (Chapter 6).

In Section 5.11, we discuss limitations and the assumptions we make for the refinement lifting of the bootstrapping phase, and in Section 5.12, we present the proofs of the aforementioned theorems and discuss their complexity.

5.7 Kernel Isolation Theorem

With regards to kernel verification, an important design aspect of the clustered multikernel is that a node's kernel is isolated from other nodes. This makes verification more tractable because we can reason about each node in isolation. It is important for part 2 of the bootstrapping phase, and even more, for the runtime phase (Chapter 6). However, we first have to prove isolation itself.

5.7.1 Definition

We define two nodes' kernels are isolated if they never access the same kernel data. In the seL4 model, this means that no two nodes' kernel objects ever overlap or are shared. At runtime, kernel objects can only be created in memory regions to which the user has an untyped-memory capability. Hence, the memory region covered by all untyped-memory capabilities is the memory region that can maximally be occupied by new kernel objects. However, we also have to take into account existing kernel objects which may not be covered by an untyped-memory capability. This is possible for kernel objects that were created during bootstrapping or whose parent untyped-memory capability was deleted after the object had been created.

For the *kernel isolation theorem*, we use the fact that during bootstrapping, kernel objects and untyped-memory capabilities are only created within a node's *non-shared memory region*, which is defined as the union of the user-level image's memory region and the available memory region provided to that node. The non-shared memory region is private to each node.

Theorem 5.7.1 (Kernel Isolation)

```
⟦node_1_id < num_nodes; node_2_id < num_nodes⟧
⟹ {|λs. s = init_glms|} boot_sys
    {|λ_ s. ∃reg_1 reg_2.
              ndks_prop (λndks. reg_1 = ndks_non_sh_p_reg ndks) node_1_id s ∧
              ndks_prop (λndks. reg_2 = ndks_non_sh_p_reg ndks) node_2_id s ∧
              ndks_prop (λndks. p_region_of_kobj_ut ndks ⊆ reg_1) node_1_id s ∧
              ndks_prop (λndks. p_region_of_kobj_ut ndks ⊆ reg_2) node_2_id s ∧
              (node_1_id ≠ node_2_id ⟶ reg_1 ∩ reg_2 = ∅)|}
```

The non-shared memory region is stored as ghost field ndks_non_sh_p_reg in each node's *ndks_t*. We will need this ghost field in Section 5.9. For the actual isolation statement here, the two clauses involving this field are not important.

The important part is the function p_region_of_kobj_ut, which returns a node's region of memory that is covered by all kernel objects and untyped-memory capabilities. It is used inside ndks_prop, which lifts a property over a node's static kernel state.

Definition 5.7.2 (Static-Node-Kernel-State Lifting)

```
ndks_prop :: (ndks_t ⇒ bool) ⇒ node_id_t ⇒ glms_t ⇒ bool
ndks_prop P node_id s
    ∃ndks. glms_pmem_var s var_ndks_list_[node_id] = ⌊VarNdKs ndks⌋ ∧ P ndks
```

This function lifts a property defined over type *ndks_t* into the *glks_t* state monad (in which e.g. boot_sys is defined). It is parameterised by the ID of the desired node's *ndks_t*.

In summary, the kernel isolation theorem states that after the system is bootstrapped, for any two distinct nodes, there exist disjoint memory regions (which are the non-shared regions) with each being a superset of its node's p_region_of_kobj_ut. This implies that all nodes' p_region_of_kobj_ut are disjoint.

Showing the proof of this theorem and discussing its complexity is deferred to the end of this chapter (Section 5.12.3).

5.7.2 Connection to the Runtime Phase

According to the seL4 API, during the runtime phase, no user input will ever be able to create kernel objects such that two distinct node's kernel objects overlap. However, while refinement proves that the implementation refines the abstract specification, it does not automatically prove that the abstract specification correctly models the desired behaviour described above. For example, there might be a subtle flaw in the specification enabling the user to create a kernel object that covers a memory region outside its parent untyped-memory capability.

Consequently, in addition to the kernel isolation theorem presented above, we need a theorem that also covers the runtime phase by stating that no kernel objects can ever be created outside the memory region covered by untyped-memory capabilities. A very similar theorem had already been proved for seL4's abstract specification during the L4.verified project. We were able to entirely reuse that theorem by slightly reformulating the affected memory region. In Section 5.9, we will describe how we formally connect these two theorems.

5.7.3 Summary

An important design aspect of the clustered multikernel is that a node's kernel is isolated from other nodes. In order to prove this, the kernel isolation theorem states that after

the system is bootstrapped, for any two distinct nodes, there exist disjoint memory regions (which are the non-shared regions) with each being a superset of its node's p_region_of_kobj_ut. This implies that all nodes' p_region_of_kobj_ut are disjoint.

We also identified that the theorem needs a counterpart covering the runtime phase of the kernel, which will be discussed in Section 5.9.

5.8 User-Level Theorems

The two user-level theorems presented in this section are not strictly necessary for seL4::CMK's kernel refinement proof. However, together with the kernel isolation theorem, the first one—the *user-level isolation theorem*—provides node-isolation guarantees which can be useful for building systems on top of seL4::CMK.

5.8.1 Definition

Theorem 5.8.1 (User-Level Isolation)

```
⟦node_1_id < num_nodes; node_2_id < num_nodes⟧
⟹ {|λs. s = init_glms|} boot_sys
    {|λ_ s. ∃reg_1 reg_2.
                ndks_prop (λndks. reg_1 = ndks_non_sh_p_reg ndks) node_1_id s ∧
                ndks_prop (λndks. reg_2 = ndks_non_sh_p_reg ndks) node_2_id s ∧
                ndks_prop (λndks. p_region_of_non_sh_frame_caps ndks ⊆ reg_1)
                 node_1_id s ∧
                ndks_prop (λndks. p_region_of_non_sh_frame_caps ndks ⊆ reg_2)
                 node_2_id s ∧
                (node_1_id ≠ node_2_id ⟶ reg_1 ∩ reg_2 = ∅)|}
```

The structure of the postcondition is the same as in the kernel isolation theorem. The difference is the function p_region_of_non_sh_frame_caps, which returns the physical memory region covered by all user-level frames that are not supposed to be shared between nodes. This includes frames that contain the user-level image and special-purpose frames such as the initial thread's *IPC buffer* and the *boot-info frame*. The IPC buffer will be introduced in Section 6.6.3. The boot-info frame is used by the kernel to provide configuration information to the initial thread.

Essentially, the theorem states that after kernel bootstrapping has finished, the user-level-image and special-purpose frames are not shared between nodes. The function p_region_of_non_sh_frame_caps identifies these frames by a ghost flag stored in their capabilities. The ghost flags are set by the abstract bootstrapping code when creating the frames.

Combined with the kernel isolation theorem (which covers kernel-accessed objects), the user-level isolation theorem guarantees that nodes are isolated as intended.

Remember that nodes are not completely isolated. There are two types of frames that can be shared between nodes. The first type is *device frames*. These are frames that allow user level to access a device's memory-mapped registers. The second type is *shared frames*, which are used at user level to implement communication and coordination between nodes.

The second user-level theorem, the *user-level sharing theorem*, talks about shared frames.

Theorem 5.8.2 (User-Level Sharing)

```
{|λs. s = init_glms|} boot_sys
{|λ_ s. ∃reg. ∀node_id<num_nodes.
                    ndks_prop (λndks. p_region_of_sh_frame_caps ndks = reg)
                     node_id s|}
```

The function p_region_of_sh_frame_caps returns the physical memory region covered by all frames provided at user level as shared frames. Hence, this theorem states that after the kernel has been bootstrapped, the shared frames provided at user level on each node are actually shared between the nodes, i.e. they are backed by the same physical memory region *reg* on all nodes.

Showing the proofs of these theorems and discussing their complexity is deferred to the end of this chapter (Section 5.12.4).

5.8.2 Summary

While the two user-level theorems presented in this section are not strictly necessary for seL4::CMK's refinement proof, they can be useful for building systems on top of seL4::CMK. For example, together with the kernel isolation theorem, the user-level isolation theorem provides isolation guarantees between nodes which applications can rely on.

5.9 Connection to L4.verified Proofs

The abstract specification of seL4::CMK's bootstrapping phase was written independently from the L4.verified project. The aim was to have a standalone specification that is independent from the seL4 specification/proof base. One of the reasons for doing this was being able to selectively abstract away from the more low-level data representation used by L4.verified. For example, the seL4::CMK bootstrapping specification uses the *nat* type to represent 32-bit numbers whereas L4.verified uses *word32*, a type that exactly models the behaviour of a machine word, such as wrapping around. However, we have to somehow connect seL4::CMK's bootstrapping specification/proofs to the L4.verified formalisation in order to be able to reason about both bootstrapping and runtime phases together, which is necessary to prove refinement.

For the moment, we are looking at nodes running on one CPU each. Enabling nodes to run on more than one CPU will be discussed in Chapter 6. When running on one CPU, an seL4::CMK node is indistinguishable from the uniprocessor version of seL4 (on the abstract level). In order to link bootstrapping and runtime phase, we therefore have to connect each bootstrapped node's *ndks_t* to the uniprocessor state of the L4.verified formalisation (type *state*).

Our proposed technique for doing this combines ghost state with a *state relation*. A state relation relates states of two different state types. For each pair of possible states of these types, it defines whether they are related or not. Formally, a state relation is a set of state pairs. If a pair is in the set, the two states it consists of are related; otherwise, they are not. Remember that in Isabelle, a set of pairs is represented as a function from pairs to *bool*. Hence, for any given pair of states, the function decides whether they are related or not.

First, we define the state relation with the following name/type:

rel_ndks_state :: *ndks_t* × *state* ⇒ *bool*

This state relation defines which states in the *ndks_t* type are related to which states in L4.verified's *state* type. The definition comprises 160 LOC. The majority of it relates data structures that are modelled equally but use different names or a different number type (*nat* vs. *word32*). Sometimes, kernel object content is modelled slightly differently, e.g. a proper list vs. a function from a list index to an element. We also need to relate all

datatype constructors that define object/capability types (and their content).

The following three definitions show excerpts from the state relation that give some insight on how the relation works.

The part of rel_ndks_state that relates the two formalisations of the kernel heap is defined as follows.

Definition 5.9.1 (Kernel-Heap Relation)

```
rel_kheap :: kheap_t ⇒ kheap ⇒ bool
rel_kheap kh kh' ≡ ∀p p'. rel_nw p p' ⟶ rel_opt (kh p) (kh' p') rel_obj
```

The predicate rel_nw evaluates whether the *nat* and *word32* numbers passed to it as arguments relate, i.e. if they represent the same number. The predicate rel_opt relates the option type: If both arguments are None, it evaluates to True. If only one is None, it returns False. If both are ⌊*x*⌋, it tests whether the *x* relate, using the function passed as its third parameter. Here, that function is rel_obj.

The part of rel_obj that relates CNode kernel objects is defined as follows.

Definition 5.9.2 (Object Relation)

```
rel_obj :: obj_t ⇒ kernel_object ⇒ bool
rel_obj (ObjCNode cn) obj' = (∃sz' cn'. obj' = CNode sz' cn' ∧ rel_obj_cnode cn cn')

rel_obj_cnode :: cnode_t ⇒ cnode_contents ⇒ bool
rel_obj_cnode cnode cnode' ≡
    ∃sb. 2^sb = length cnode ∧
         (∀i'∈dom cnode'. length i' = sb) ∧
         (∀i i'. i < length cnode ∧ rel_nw i (of_bl i') ⟶
                 rel_cap cnode_[i] (the (cnode' i')))
```

For CNodes, we relate their sizes (first two clauses) and their contents (last clause). For the latter, we relate every capability in the CNode, which is done by the predicate rel_cap. Note that dom and of_bl are necessary because in L4.verified, the index into a CNode is stored as binary number in a list of *bool*. In our formalisation, the index is a simple *nat*.

The part of rel_cap that relates untyped-memory capabilities is defined as follows.

Definition 5.9.3 (Untyped-Memory-Capability Relation)

```
rel_cap :: cap_t ⇒ cap ⇒ bool
rel_cap (CapUT pptr sb) c' =
    ∃oref' fi'. c' = UntypedCap oref' sb fi' ∧ rel_nw pptr oref'
```

While the state relation connects the two formalisations, it does not connect the proofs automatically. In order to do that, we have to formulate the desired theorem about the bootstrapped state in the L4.verified formalisation, using the state relation, and then prove it. To this end, we define the following *extraction function*, which already contains the state relation.

Definition 5.9.4 (Extraction Function)

```
boot_sys_A :: node_id_t ⇒ state ⇒ bool
boot_sys_A node_id ≡
    rel_ndks_state `` {ndks | ∃glms∈snd ` fst (boot_sys init_glms).
    glms_pmem_var glms var_ndks_list_[node_id] = ⌊VarNdKs ndks⌋}
```

This function extracts the non-deterministic set of initial states (after bootstrapping) of the node with ID `node_id` and "converts" them into the L4.verified formalisation via the state relation rel_ndks_state.

The extraction function can now be used to formulate theorems about the bootstrapped state in the L4.verified formalisation. We use it in the following section and for the *multikernel refinement theorem* later in Section 5.10.

5.9.1 Node Isolation at Runtime

In Section 5.7, we mentioned that in addition to the kernel isolation theorem presented there, we need a theorem that also covers the runtime phase by stating that no kernel objects can ever be created outside the memory region covered by untyped-memory capabilities.

Coincidentally, L4.verified proved an invariant that we were able to completely reuse for our purpose. The invariant states that the memory region covered by all kernel objects, untyped-memory capabilities and user-level frames will never grow. Originally, it was indented to prove that no kernel object or user-level frame is ever allocated outside the *kernel window*.[10] To this end, the memory region was defined as a ghost field in the `state` record. This field never gets changed during runtime, it is supposed to be initialised during bootstrapping. However, L4.verified did not provide a bootstrapping specification. Therefore, we are free reinterpret this field: We define the field to contain the node's non-shared memory region.

Here is where the aforementioned ghost state comes into play: Remember that in the seL4::CMK specification, the non-shared memory region is stored in the ghost field ndks_non_sh_p_reg of the `ndks_t` record. The kernel and user-level isolation theorems state that the memory region covered by all kernel objects, untyped-memory capabilities and non-shared frames is a subset of ndks_non_sh_p_reg. We can now link the notion of the non-shared memory region of both formalisation by relating the ghost field ndks_non_sh_p_reg to its L4.verified counterpart. This relation is part of the state relation rel_ndks_state.

So far, we have proved that bootstrapping establishes kernel isolation between nodes and that it is preserved at runtime. The former is proved in the seL4::CMK formalisation whereas the latter is proved in the L4.verified formalisation. We have also defined how these formalisations relate to each other. The relation covers the real state and the ghost state. Hence, the only missing link is a proof connecting the two formalisations of node isolation and therefore the proofs of establishment and preservation of node isolation.

To this end, we formulate—in the L4.verified formalisation—the *node isolation theorem* stating that seL4::CMK's bootstrapping establishes that, for each node, the memory region covered by all kernel objects, untyped-memory capabilities and non-shared frames lies within the non-shared memory region.

Theorem 5.9.5 (Node Isolation)

$$\frac{\texttt{node\_id} < \text{num\_nodes} \qquad \text{valid\_pspace } s \qquad s \in \text{boot\_sys\_A } \texttt{node\_id}}{\text{pspace\_in\_kernel\_window } s \wedge \text{cap\_refs\_in\_kernel\_window } s}$$

The state `s` is the node's state after bootstrapping, in the L4.verified formalisation. The conjunction in the goal of this theorem states that the region covered by all kernel objects, untyped-memory capabilities and non-shared frames lies within the kernel window, i.e. the ghost field in `s` that stores the kernel-window memory region. Remember that this

[10] The *kernel window* is a region of virtual memory that maps to the physical memory occupied by the kernel heap. For performance reasons, the kernel accesses the kernel heap via this kernel window.

ghost field is related to the field ndks_non_sh_p_reg in the seL4::CMK formalisation of a node's state though the state relation rel_ndks_state (which appears in boot_sys_A).

In the assumptions, we use the function boot_sys_A (Definition 5.9.4) to extract the `ndks_t` state of the node with ID `node_id`. The property valid_pspace is an L4.verified invariant that ensures a valid kernel heap.

With this, we have proved that L4.verified's invariants pspace_in_kernel_window and cap_refs_in_kernel_window are established by seL4::CMK's bootstrapping, with the kernel window being interpreted as the node's non-shared memory region. L4.verified has already proved that these two invariants hold during the runtime phase.

As a consequence, we have a connected proof of establishment and preservation of node isolation.

5.9.2 Proof

For proving the node isolation theorem, we first unfold the definitions of boot_sys_A and the state relation rel_ndks_state in boot_sys_A. This gives us access to boot_sys and allows us to insert the kernel and user-level isolation theorems. Next, we unfold pspace_in_kernel_window and cap_refs_in_kernel_window, which are the L4.verified formalisation of containment in the non-shared memory region.

With both sides unfolded, we do a case distinction over 8 object types and 16 capability types. For each type, we prove that containment in the seL4::CMK formalisation, converted to the L4.verified formalisation via the state relation, implies containment as defined in the L4.verified formalisation.

■

5.9.3 Discussion

The proof required 230 LOC. The complexity is distributed unevenly. Certain object and capability types were trivial to prove, others were more complex. An example of the latter is the CNode object type. It is complex for two reasons: First, it stores capabilities, which means that object type and capability type case distinctions are nested. Second, it is represented differently in the two formalisations: In seL4::CMK, the list of capabilities is represented directly as Isabelle list. In L4.verified, it is represented as a function from list index to capability. The index number itself is a bit list that is the binary representation of the index number.

However, a surprisingly high share of the 230 LOC were required for conversions between the `nat` and `word32` number types. This is the result of Isabelle providing low automation for conversion between these number types. As this is not specific to the node isolation theorem, we would expect this problem to also arise in further proofs involving the state relation rel_ndks_state.

Hence, we find that it is not advisable to use different number types in formalisations that will eventually be connected via a state relation, unless a considerable degree of automation is provided for number type conversions.

5.9.4 Summary

We connected seL4::CMK's bootstrapping specification/proofs to the L4.verified formalisation of the runtime phase in order to be able to reason about both bootstrapping and runtime phases together. To achieve this, we used a technique that combines ghost state with a state relation.

This enabled a proof of establishment and preservation of node isolation that is connected between the formalisations of the bootstrapping phase and the runtime phase.

We showed that our technique is a viable way of connecting two different formalisations of the same system, as long as the same number type is chosen.

5.10 Multikernel Refinement Theorem

With node isolation proved, we are now able to tackle the first top-level refinement theorem.

In this section, we are only looking at nodes running on one CPU, i.e. a *multikernel* (without clustering). This is an intermediate step towards the final *clustered-multikernel refinement theorem* of Chapter 6, which will address multiple CPUs per node. While the multikernel refinement theorem itself is of less importance, the definitions and lemmas presented in this section will be required by Chapter 6.

First, we define seL4's multikernel ADTs. The abstract multikernel ADT combines seL4's abstract specification of the runtime phase with seL4::CMK's abstract bootstrapping specification. The concrete multikernel ADT includes seL4's intermediate specification of the runtime phase and axiomatises a concrete seL4::CMK bootstrapping specification, which is left for future work (see Section 5.11.3). We base our definitions and theorems on the refinement calculus described in Section 2.2 and its application to uniprocessor seL4 (Section 2.6).

We start defining the Init functions of the ADTs.[11]

Definition 5.10.1 (Multikernel ADT Init Functions)

```
orig_init_A :: node_id_t ⇒ state global_state ⇒ bool
orig_init_A node_id ≡ ({empty_context} × boot_sys_A node_id) × {UserMode} × {None}

orig_init_C :: node_id_t ⇒ kernel_state global_state ⇒ bool
orig_init_C node_id ≡ ({empty_context} × boot_sys_C node_id) × {UserMode} × {None}
```

On both abstract and concrete levels, we initialise a node to start running in UserMode with None events pending and an empty_context for the CPU registers. The initial kernel state is returned by boot_sys_A (Definition 5.9.4) for the abstract level and by boot_sys_C for the concrete level. Due to the missing concrete bootstrapping specification, boot_sys_C is not defined, only declared.

Definition 5.10.2 (Multikernel ADTs)

```
orig_ADT_A :: node_id_t ⇒ (state global_state, observable,
              global_transition) adt_t
orig_ADT_A node_id ≡ ADT_A(|Init := orig_init_A node_id|)

orig_ADT_C :: node_id_t ⇒ (kernel_state global_state, observable,
              global_transition) adt_t
orig_ADT_C node_id ≡ ADT_C(|Init := orig_init_C node_id|)
```

ADT_A is uniprocessor seL4's abstract ADT and ADT_C is its concrete ADT, i.e. the ADT of the intermediate specification. The multikernel ADTs inherit the Step and Fin function of seL4's ADTs but replace the Init function.

[11] We prefix all constants related to the multikernel ADTs with ***orig**inal*. We do this because these ADTs will be the *original ADTs* in Chapter 6 when taking the step from single-CPU nodes to multiple-CPU nodes for the runtime phase.

Before being able to prove refinement, we have to prove that the invariants hold on both abstract and concrete levels.

Lemma 5.10.3 *The invariants hold for both abstract and concrete multikernel ADTs.*

$$node\_id < \text{num\_nodes} \implies \text{orig\_ADT\_A}\ node\_id \models \text{full\_invs}$$
$$node\_id < \text{num\_nodes} \implies \text{orig\_ADT\_C}\ node\_id \models \text{full\_invs}'$$

In order to prove this lemma, we need two axioms. We have already mentioned the first of them: There is no concrete seL4::CMK bootstrapping specification (boot_sys_C), therefore we have to axiomatise the invariants it is supposed to establish. A small subset of invariants do not have to be axiomatised because they can be inferred from the definition of orig_init_C.

The second axiom establishes a subset of the abstract invariants. In Section 5.9, we proved that seL4::CMK's abstract bootstrapping specification establishes the invariants pspace_in_kernel_window and cap_refs_in_kernel_window. Hence, these two invariants can be taken out of the axiomatisation, together with the invariants that can be inferred from orig_init_A. However, we axiomatise the remaining invariants because proving them is not in the scope of this work.

In order to prove refinement, we prove forward simulation first.

Lemma 5.10.4 *For each node,* orig_ADT_C *forward-simulates* orig_ADT_A *(if the invariants hold).*

$$\frac{node\_id < \text{num\_nodes}}{\text{fw\_sim\_inv}\ (\text{orig\_ADT\_A}\ node\_id)\ (\text{orig\_ADT\_C}\ node\_id)\ \text{refine\_rel}\ (\text{full\_invs} \times \text{full\_invs}')}$$

Note that refine_rel is the L4.verified refinement relation.
Proving this lemma requires us to prove correspondence of the Init, Step and Fin functions of the ADTs. This is depicted in Figure 5.3 with the example of two nodes. Step and Fin are already proved by L4.verified. However, due to the missing intermediate seL4::CMK bootstrapping specification, we axiomatise correspondence of most of Init, i.e. up to the extraction functions.

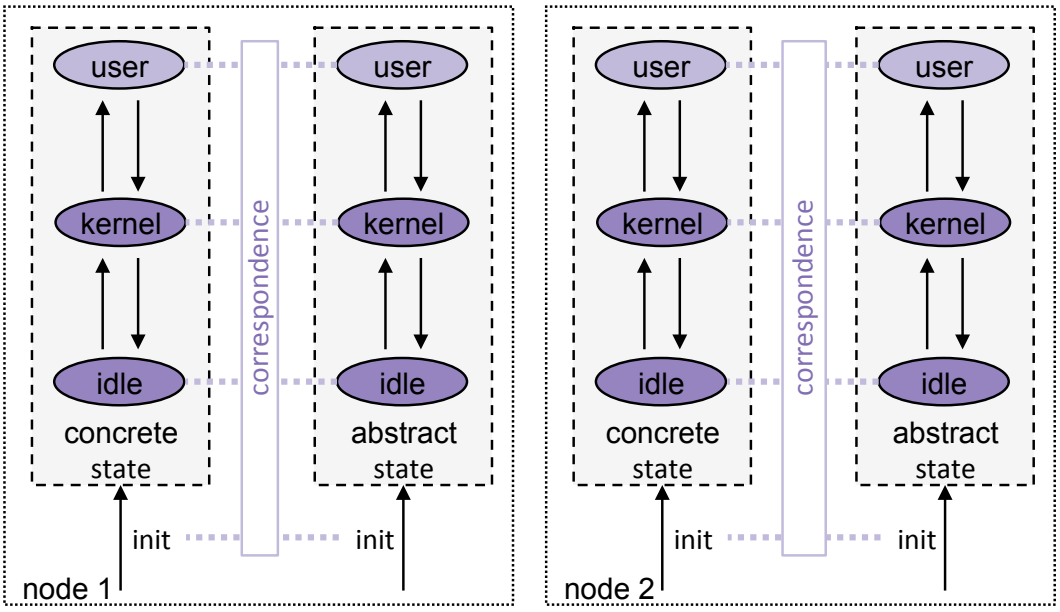

Figure 5.3: *Required Correspondence Proofs in the Multikernel Refinement (two nodes with one CPU each are depicted)*

Axiom 5.10.5 *The extraction functions* boot_sys_A *and* boot_sys_C *correspond.*

$$\frac{\texttt{node\_id} < \text{num\_nodes}}{\text{boot\_sys\_C } \texttt{node\_id} \subseteq \text{state\_relation } \texttt{''} \text{ boot\_sys\_A } \texttt{node\_id}}$$

Note that state_relation is the part of refine_rel that relates the `state` record of the abstract specification to the `kernel_state` record of the intermediate specification.

Using Lemma 5.10.3 and Lemma 5.10.4 in the assumptions of Theorem 2.2.9 yields the following lemma.

Lemma 5.10.6 *For each node,* orig_ADT_C *forward-simulates* orig_ADT_A.

$$\frac{\texttt{node\_id} < \text{num\_nodes}}{\text{orig\_ADT\_C } \texttt{node\_id} \sqsubseteq_F \text{orig\_ADT\_A } \texttt{node\_id}}$$

After applying Theorem 2.2.6, we get the final multikernel refinement theorem.

Theorem 5.10.7 (Multikernel Refinement) *For each node,* orig_ADT_C *refines* orig_ADT_A.

$$\frac{\texttt{node\_id} < \text{num\_nodes}}{\text{orig\_ADT\_C } \texttt{node\_id} \sqsubseteq \text{orig\_ADT\_A } \texttt{node\_id}}$$

As mentioned before, the theorem itself is not strictly required by Chapter 6, where we address the step to nodes spanning multiple CPUs. However, the definitions and lemmas in this section leading to this theorem are required. More specifically, Lemma 5.10.3 and Lemma 5.10.4 will be the key lemmas required in Chapter 6.

5.11 Assumptions and Limitations

5.11.1 Compiler/Boot Loader

We assume correctness of compiler and boot loader. More precisely, we axiomatise certain properties about the memory layout they generate. We also axiomatise that the boot loader provides sane system configuration data, e.g. that each CPU ID only exists once. Details are discussed in Section 5.4.1.

5.11.2 Overapproximation

As described in Section 5.4, we overapproximate memory accesses in the abstract specification. This means that for abstract read/write operations, we do not specify in which order, for example, a kernel object's memory addresses are read/written by the final assembly code.

We use overapproximation for two reasons: (1) Different ways of implementing these operations in C result in different access patterns and (2) different C compilers emit different assembly code which also results in varying patterns. For example, the compiler might optimise away or reorder memory accesses. It is also undefined in which order a C compiler evaluates expressions.

We assume that whenever the C implementation accesses a kernel object, it only accesses memory addresses within the kernel object's memory region. This assumption is reasonable as it should hold in every sensible C implementation. Nevertheless, the problem of bugs in the C implementation violating this assumption remains. However, it has to be noted that a large subclass of these bugs, wild writes, would have been caught already by the normal refinement proof.

It would be possible to remove this assumption by also instrumenting the intermediate specification and the C parser to generate an abstract parallel program. The refinement relation would have to define what correspondence between two such parallel programs means, and the refinement proof would have to prove that they correspond.

5.11.3 Missing Correspondence Proofs

Recall from Section 2.6.1 that bootstrapping was not in the scope of the L4.verified project. The seL4 refinement proof axiomatises correct bootstrapping (correspondence and establishment of invariants). Therefore, we are not able to leverage an existing bootstrapping proof as intended by the refinement lifting framework.

As a consequence, we axiomatise most of the invariant and correspondence proofs about seL4::CMK's bootstrapping and leave them for future work. We concentrate on proving theorems that are important with regards to concurrency, such as the sequential-semantics and isolation theorems.

However, in the meantime, an abstract and an intermediate specification have been written for uniprocessor seL4's bootstrapping, based on the abstract specification of seL4::CMK's bootstrapping presented in this thesis. The invariant and correspondence proofs to remove the uniprocessor bootstrapping axioms are currently being worked on by the seL4 verification team. When finished, they need only be slightly adapted to be applicable to part 2 of seL4::CMK's bootstrapping. Part 1 of bootstrapping needs its own invariant and correspondence proofs.

These correspondence proofs can be carried out in the same way and with the same verification framework they had been carried out in seL4's past. There is nothing concurrency-specific in them.

5.12 Proofs

In this section, we present the proofs that have been deferred in the previous sections.

5.12.1 Sequential-Semantics Theorem

This section presents the proof of the sequential-semantics theorem, which we introduced in Section 5.3. We present it in a top-down manner, and therefore start with the theorem itself.

Theorem (Sequential Semantics)

$$\frac{\mathsf{instr\_mfence\_start\_cpu}\ il \qquad \mathsf{instr\_rw\_p\_regs\_disj}\ il}{\mathsf{observes\_seq\_sem}\ c\ il}$$

The design of the proof is based on applying the following lemma to the transition relation trans_rel.

Lemma 5.12.1 (Invariance in a Reflexive Transitive Closure)

$$\bigwedge x\ y.\ \frac{(x,\ y) \in R \qquad I\ x}{I\ y} \qquad (s,\ s') \in R^* \qquad I\ s \;\Longrightarrow\; I\ s'$$

For any pair of start and end state s and s' in the reflexive transitive closure of transition relation R, the invariants I are preserved if they are preserved by every transition x to y in the relation. Note that the Isabelle syntax $\bigwedge x\ y.$ means "for all x and y".

The idea is to find invariants inv that (1) hold in the initial state init_mpss *c il*, (2) are preserved by every transition in the transition relation trans_rel and (3) imply the goal observes_seq_sem *c il*. To this end, we define the following three lemmas.

Lemma 5.12.2 *The invariants hold in the initial state of the system.*

$$\frac{\text{instr\_mfence\_start\_cpu } il \qquad \text{instr\_rw\_p\_regs\_disj } il}{\text{inv } (\text{init\_mpss } c\ il)}$$

Lemma 5.12.3 *The invariants are preserved by every transition defined in* trans_rel.

$$\frac{(s,\ s') \in \text{trans\_rel}}{\text{inv } s \longrightarrow \text{inv } s'}$$

Lemma 5.12.4 *The invariants imply the sequential-semantics property.*

$$\frac{\text{inv } s}{\text{al\_has\_seq\_sem } (\text{mpss\_al } s)}$$

We have to define invariants inv such that the three lemmas above are true. The lemmas then imply the sequential-semantics theorem with the help of Lemma 5.12.1.

Finding a suitable set of invariants is not straight-forward. It took several iterations of invariant fine-tuning and proof attempts until we were able to finish the proof. The problem is two-fold: First, the assumptions and goal of the sequential-semantics theorem are complex enough that it is not immediately clear by looking at invariant candidates whether the three lemmas above are true or not. Often, corner cases falsifying a lemma would only become visible half-way into a proof attempt. Second, even if the chosen set of invariants satisfied the lemmas, the way they were formulated increased proof complexity such that it was wiser to give up the proof and reformulate the invariants instead of trying to finish the proof.

The following six invariants led to a successful proof.

Definition 5.12.5 (Invariants)

```
inv :: mpss_t ⇒ bool
inv s ≡ inv_mfence s ∧ inv_disj s ∧ inv_rw s ∧ inv_cr s ∧ inv_al s ∧
        al_has_seq_sem (mpss_al s)
```

The last invariant is trivial, it is exactly the sequential-semantics property. The next five definitions cover the remaining invariants.

Definition 5.12.6 (Memory-Fence Invariant)

```
inv_mfence :: mpss_t ⇒ bool
inv_mfence s ≡
    ∀ c. case mpss_cil s c of [] ⇒ True
        | InstrMFENCE · il ⇒ instr_mfence_start_cpu (InstrMFENCE · il)
        | InstrStartCPU x start_il · il ⇒
            csb_empty c s ∧
            instr_mfence_start_cpu il ∧ instr_mfence_start_cpu start_il
        | _ · il ⇒ instr_mfence_start_cpu il
```

For every CPU, the program to be executed satisfies instr_mfence_start_cpu (correct memory-fence placement), with the following exception: If the next instruction to be executed is an InstrStartCPU, then the store buffer is empty and both the remaining program to be executed (*il*) and the new CPU's program (*start_il*) satisfy instr_mfence_start_cpu.

Definition 5.12.7 (Disjoint-Read/Write-Region Invariant)

```
inv_disj :: mpss_t ⇒ bool
inv_disj s ≡ ∀c. instr_rw_p_regs_disj (mpss_cil s c)
```

For every CPU, the remaining program to be executed satisfies the disjoint-read/write-region property.

Definition 5.12.8 (Distinct-CPU-Read/Write-Region Invariant)

```
inv_rw :: mpss_t ⇒ bool
inv_rw s ≡ ∀p c1 c2.
             c1 ≠ c2 ∧ p ∈ instr_read_p_reg (mpss_cil s c1) ⟶
             p ∉ instr_write_p_reg (mpss_cil s c2) ∪ ⋃set (mpss_csb s c2)
```

For all distinct CPUs $c1$ and $c2$, if a memory address p occurs in any InstrRead of $c1$'s program, it neither occurs in an InstrWrite of $c2$'s program nor in $c2$'s store buffer.

Definition 5.12.9 (CPU-Running Invariant)

```
inv_cr :: mpss_t ⇒ bool
inv_cr s ≡ ∀c. ¬ mpss_cr s c ⟶
               mpss_cil s c = [] ∧
               mpss_csb s c = [] ∧
               (∀a∈set (mpss_al s). cpu_of_acc a ≠ c)
```

Unless a CPU has been started, its program and store buffer are empty, and the memory-access scenario does not contain any access performed by that CPU.

Definition 5.12.10 (Memory-Access-Scenario Invariant)

```
inv_al :: mpss_t ⇒ bool
inv_al s ≡ ∀p c i.
           i < length (al_of_paddr p (mpss_al s)) ∧
           cpu_of_acc (al_of_paddr p (mpss_al s))[i] = c ∧
           p ∈ instr_read_p_reg (mpss_cil s c) ⟶
           ¬ (∃k>i. k < length (al_of_paddr p (mpss_al s)) ∧
                    cpu_of_acc (al_of_paddr p (mpss_al s))[k] ≠ c ∧
                    ¬ is_acc_read (al_of_paddr p (mpss_al s))[k])
```

For each CPU c and memory address p, if CPU c's program contains an InstrRead to memory address p, then the memory-access scenario contains no InstrWrite to memory address p by another CPU after any of CPU c's accesses.

With these invariants, we were able to conclude the sequential-semantics theorem (Theorem 5.3.6) from Lemma 5.12.2, Lemma 5.12.3, Lemma 5.12.4 and Lemma 5.12.1 with a trivial proof. Proving Lemma 5.12.2 and Lemma 5.12.4 was trivial as well. As it turned out, almost the entire complexity lies in Lemma 5.12.3. In order to prove it, we divide it into several sub-lemmas, one for each transition rule.

Lemma 5.12.11 *The invariants are preserved when executing an* InstrRead *leads to the value being read from the memory subsystem.*

$$\frac{\text{inv } s \qquad s' = \text{exec\_read } r\ c\ (\text{cil\_dequeue } c\ s) \qquad \text{cil\_next } (\text{InstrRead } r)\ c\ s \qquad \neg\ \text{csb\_contains } r\ c\ s}{\text{inv } s'}$$

This lemma corresponds to the transition rule from Definition 5.2.8. Proving it required 110 LOC of proof script.

Lemma 5.12.12 *The invariants are preserved when executing an* InstrRead *leads to the value being read from the store buffer (store-buffer forwarding).*

$$\frac{\text{inv } s \qquad s' = \text{cil\_dequeue } c\ s \qquad \text{cil\_next } (\text{InstrRead } r)\ c\ s \qquad \text{csb\_contains } r\ c\ s}{\text{inv } s'}$$

This lemma corresponds to the transition rule from Definition 5.2.9. Proving it required 40 LOC of proof script.

Lemma 5.12.13 *The invariants are preserved when an* InstrWrite *is executed.*

$$\frac{\text{inv } s \qquad s' = \text{csb\_enqueue } r\ c\ (\text{cil\_dequeue } c\ s) \qquad \text{cil\_next } (\text{InstrWrite } r)\ c\ s}{\text{inv } s'}$$

This lemma corresponds to the transition rule from Definition 5.2.10. Proving it required 60 LOC of proof script.

Lemma 5.12.14 *The invariants are preserved when a store buffer writes a value to the memory subsystem.*

$$\frac{\text{inv } s \qquad s' = \text{exec\_write } r\ c\ (\text{csb\_dequeue } c\ s) \qquad \text{csb\_next } r\ c\ s}{\text{inv } s'}$$

This lemma corresponds to the transition rule from Definition 5.2.11. Proving it required 70 LOC of proof script.

Lemma 5.12.15 *The invariants are preserved when a memory fence is executed.*

$$\frac{\text{inv } s \qquad s' = \text{cil\_dequeue } c\ s \qquad \text{cil\_next InstrMFENCE } c\ s \qquad \text{csb\_empty } c\ s}{\text{inv } s'}$$

This lemma corresponds to the transition rule from Definition 5.2.12. Proving it required 50 LOC of proof script.

Lemma 5.12.16 *The invariants are preserved when starting a CPU.*

$$\frac{\begin{array}{l}\text{cil\_next } (\text{InstrStartCPU } cn\ start\_il)\ c\ s \\ \text{inv } s \qquad s' = \text{exec\_start\_cpu } cn\ start\_il\ (\text{cil\_dequeue } c\ s)\end{array}}{\text{inv } s'}$$

This lemma corresponds to the transition rule from Definition 5.2.13. Proving it required 270 LOC of proof script.

■

Discussion

Starting a new CPU requires the most complex invariant proof. The other transitions are considerably less complex to prove.

Table 5.3 shows a breakdown by individual invariants of the 600 LOC of proof script required to prove the above six lemmas (and thereby Lemma 5.12.3).

inv_mfence	inv_disj	inv_rw	inv_cr	inv_al	al_has_seq_sem	Total
180 LOC	30 LOC	190 LOC	20 LOC	120 LOC	60 LOC	600 LOC

Table 5.3: *Proof Size for Individual Invariants of Lemma 5.12.3*

Note that these numbers represent dense proof script in Isabelle's apply-style. In order to calculate the entire proof size, we have to add 150 LOC for several small helper lemmas,

Lemma 5.12.2, Lemma 5.12.4, and for combining all lemmas. The invariant definitions take another 50 LOC. This makes 800 LOC in total for the proof of the sequential-semantics theorem.

While a certain degree of proof complexity was to be expected, we realised that complexity was further increased by the fact that the assumptions and the goal of the theorem are formulated in different styles: The assumptions are defined in an inductive and recursive way whereas the goal is formulated as a large single term. We conjecture that proof complexity can be reduced by adapting the definitions of the assumptions and the goal accordingly. While this insight may be useful for new theorems to be proved, it does not help improving the sequential-semantics theorem due to its proof already being done.

As expected, and opposed to a model-checking approach, proof complexity does not depend on the maximum number of CPUs, size of memory, size of store buffers or whether they are FIFO or not. In contrast, complexity depends on how the TSO model and the theorem are formulated and on the choice of the invariants. However, while optimal formulation of model/theorem/invariants might minimise direct proof complexity, the process of finding the optimal formulation indirectly adds considerable effort that can be higher than the saved direct proof effort.

5.12.2 Kernel-Memory-Sequential-Access Theorem

This section presents the proof of the kernel-memory-sequential-access theorem, which we introduced in Section 5.5. We present it in a top-down manner, and therefore start with the theorem itself.

Theorem (Kernel Memory Sequential Access)

$$\{\!|\lambda s.\ s = \text{init\_glms}|\!\}\ \text{boot\_sys}\ \{\!|\lambda\_\ s.\ \text{observes\_seq\_sem}\ \text{cpu\_list}_{[0][0]}\ (\text{glms\_il}\ s)|\!\}$$

First, we apply the sequential-semantics theorem backwards. Its two assumptions leave us with the following two lemmas to prove.

Lemma 5.12.17 *The bootstrapping abstract parallel program contains an* InstrMFENCE *in front of every* InstrStartCPU.

$$\{\!|\lambda s.\ s = \text{init\_glms}|\!\}\ \text{boot\_sys}\ \{\!|\lambda\_\ s.\ \text{instr\_mfence\_start\_cpu}\ (\text{glms\_il}\ s)|\!\}$$

Lemma 5.12.18 *The bootstrapping abstract parallel program satisfies the disjoint-read/write-region property.*

$$\{\!|\lambda s.\ s = \text{init\_glms}|\!\}\ \text{boot\_sys}\ \{\!|\lambda\_\ s.\ \text{instr\_rw\_p\_regs\_disj}\ (\text{glms\_il}\ s)|\!\}$$

The first proof design element is the following property, or more specifically, its negation, which we call the *no-start-CPU property*.

Definition 5.12.19 *There is at least one* InstrStartCPU *in the instruction list* `il`.

$$\text{instr\_start\_cpu\_in\_il} :: \mathit{instr\_t\ list} \Rightarrow \mathit{bool}$$
$$\text{instr\_start\_cpu\_in\_il}\ \mathit{il} \equiv \exists \mathit{instr} \in \text{set}\ \mathit{il}.\ \text{is\_instr\_start\_cpu}\ \mathit{instr}$$

Note that the predicate is_instr_start_cpu returns True if the instruction passed as argument is an InstrStartCPU, otherwise, it returns False.

Lemma 5.12.20 *The no-start-CPU property trivially implies the* instr_mfence_start_cpu *and* instr_rw_p_regs_disj *properties.*

$\neg$ instr_start_cpu_in_il *il* $\Longrightarrow$ instr_mfence_start_cpu *il*
$\neg$ instr_start_cpu_in_il *il* $\Longrightarrow$ instr_rw_p_regs_disj *il*

Like this, we can simplify the Hoare triples of all functions that are executed before the first call to start_cpu (where we append an InstrMFENCE and an InstrStartCPU). This means that we only need one Hoare triple per function and the pre- and postconditions of these Hoare triples only contain the very simple no-start-CPU property.

The first function (called by boot_sys) to contain a call to start_cpu is boot_node. Hence, this function needs *transition Hoare triples*, i.e. Hoare triples with a weaker postcondition than precondition. Specifically, boot_node has two transition Hoare triples: While both of them have the no-start-CPU property in their precondition, one Hoare triple's postcondition is instr_mfence_start_cpu while the other is instr_rw_p_regs_disj. Therefore, the proof complexity lies in boot_node, where a node's remaining CPUs are started, and at the end of boot_sys, where each node's first CPU is started.

Proving instr_mfence_start_cpu Hoare triples is straight-forward. The only function that appends the relevant instructions (InstrMFENCE and InstrStartCPU) is start_cpu. Neither of these instructions is appended in any other function. Within start_cpu, the two instructions are appended at the start and the end, respectively. It is relatively simple to prove that no other instructions are appended in between.

The disjoint-read/write-region property (instr_rw_p_regs_disj, Definition 5.3.3) is much more complex to prove. It is recursive with regards to itself and it relies on instr_read_p_reg and instr_write_p_reg, which are recursive themselves. Recursive list-based definitions like this facilitate reasoning about programs that *prepend* list elements. In our case, that would be prepending abstract instructions to an existing (initially empty) parallel program. However, the abstract specification does the opposite: It *appends* abstract instructions. This prevents a straight-forward inductive proof. Instead, we have to define new properties which "remember" what we have appended so far. This knowledge is necessary in order to know if appending a particular abstract instruction violates the previously satisfies instr_rw_p_regs_disj property. Hence, the main proof design is based on what we call the *start read region* and *start write region* of an abstract parallel program.

Definition 5.12.21 (Start Read Region)

```
instr_read_p_reg_start :: instr_t list ⇒ p_region_t
instr_read_p_reg_start [] = ∅
instr_read_p_reg_start (InstrRead _ · il) = instr_read_p_reg_start il
instr_read_p_reg_start (InstrWrite _ · il) = instr_read_p_reg_start il
instr_read_p_reg_start (InstrMFENCE · il) = instr_read_p_reg_start il
instr_read_p_reg_start (InstrStartCPU _ start_il · il) =
    instr_read_p_reg_start il ∪ instr_read_p_reg start_il
```

This function returns the union of the read regions (instr_read_p_reg, Definition 5.3.4) of all programs *start_il* that are stored in InstrStartCPU instructions. The start write region (function instr_write_p_reg_start) is defined accordingly.

The start read region of an abstract parallel program is almost identical to the read region, except that the region does not contain the reads of the main CPU, i.e. the CPU that starts executing the program. In the example depicted on the left side of Figure 5.2, this is "CPU 0". The same applies to the start write region. This small difference is essential: Now we are in the position to decide if appending a particular instruction results in a conflict

between the read/write regions of what we append and of the programs already started on other CPUs. This enables an inductive proof where the induction step is *appending* instead of *prepending*.

The complexity of proving the instr_rw_p_regs_disj property is concentrated at the end of boot_sys, where we loop over all remaining nodes and start each node's first CPU. In each loop iteration, we append an InstrStartCPU which contains the abstract parallel program `start_il` to start that particular node. For each iteration, we have to prove that the *read region* of `start_il` is disjoint from the *start write region* of the already accumulated program, and accordingly for the *write region* of the former and the *start read region* of the latter. From this, it follows that instr_rw_p_regs_disj still holds after having appended the InstrStartCPU instruction.

The proof of disjoint read/write regions in boot_sys is complex because the regions include every memory address a node's bootstrapping potentially accesses. For example, the write region of boot_node comprises: the node's static kernel state (ndks_p_reg region), and the node's accessed kernel-heap memory region. The read region additionally contains the global kernel state (glks_p_reg region), where the configuration is read from. In each loop iteration, we have to relate the read and write regions of the currently starting node with the ones of the already started nodes.

At the end of boot_node, we have a second (nested) loop, which starts the remaining CPUs of the node. The same way of reasoning applies here. Nevertheless, the read/write regions are trivial: The started CPUs only read from the global kernel state (glks_p_reg region) in order to get the necessary information to configure themselves.

Appending InstrRead and InstrWrite is simpler, because in our case, they are all appended before the first InstrStartCPU. The proof goals are trivial because the start read/write regions are empty at the time of appending.

Appending an InstrMFENCE is trivial because it contains no memory access.

■

Discussion

Table 5.4 lists the LOC necessary to prove the kernel-memory-sequential-access theorem.

Proof Size	General Lemmas	Theorem-Specific	Total
Bootstrapping Part 1	1230 LOC	2080 LOC	3310 LOC
Bootstrapping Part 2	1330 LOC	360 LOC	1690 LOC
Total	2560 LOC	2440 LOC	5000 LOC

Table 5.4: *The Kernel-Memory-Sequential-Access Theorem's Proof Size*

Under *Theorem-Specific* we list the proof effort that is specific to the kernel-memory-sequential-access theorem, i.e. the theorem itself and all lemmas that are only used for proving this theorem. In contrast, the numbers under *General Lemmas* comprise proofs of lemmas that are also used by the theorems presented in the next sections. Almost all lemmas included in these numbers are Hoare triples.

We can observe that in bootstrapping part 1, the general lemmas require considerably less proof effort than the theorem-specific ones, whereas the opposite is true for part 2. The high number of LOC for part 2's general lemmas stems from its more complex data structures and from it performing dynamic allocation. For example, in order for the read_obj and write_obj operations to be safe, we have to prove that certain invariants on objects are established at a specific time in bootstrapping and are preserved until bootstrapping

finishes. Examples of such invariants are type stability and object-type-specific validity checks. In comparison, the general lemmas of bootstrapping part 1 are mostly concerned about accessing instances of `glks_t` and `ndks_t`. These types are less complex than the kernel object types of part 2, and they are statically allocated.

A considerable 330 LOC of the 1230 LOC of general lemmas in bootstrapping part 1 were required to prove that split_region splits up the total available memory between the nodes correctly. Remember that this function implements the splitting algorithm on the same level of detail as the C implementation, i.e. the function definitions are a line-by-line match.

On the theorem-specific side, most complexity lies in part 1. The reason is that in part 2, we only collect the kernel-heap accesses, whereas the actual appending of abstract instructions is done in part 1. Further complexity is added to part 1 by the prepend/append discrepancy mentioned earlier.

Note that while the kernel-memory-sequential-access theorem's proof itself is specific to seL4::CMK, the proof design and ideas are not. They could also be used for and applied to other clustered multikernels.

5.12.3 Kernel Isolation Theorem

This section presents the proof of the kernel isolation theorem, which we introduced in Section 5.7.

Theorem (Kernel Isolation)

```
⟦node_1_id < num_nodes; node_2_id < num_nodes⟧
⟹ {|λs. s = init_glms|} boot_sys
    {|λ_ s. ∃reg_1 reg_2.
              ndks_prop (λndks. reg_1 = ndks_non_sh_p_reg ndks) node_1_id s ∧
              ndks_prop (λndks. reg_2 = ndks_non_sh_p_reg ndks) node_2_id s ∧
              ndks_prop (λndks. p_region_of_kobj_ut ndks ⊆ reg_1) node_1_id s ∧
              ndks_prop (λndks. p_region_of_kobj_ut ndks ⊆ reg_2) node_2_id s ∧
              (node_1_id ≠ node_2_id ⟶ reg_1 ∩ reg_2 = ∅)|}
```

The theorem essentially makes two statements: (1) Containment: Within each node, the memory region covered by all kernel objects and untyped-memory capabilities is a subset of the non-shared region. (2) Disjunction: All nodes' non-shared regions are disjoint.

The postcondition of the theorem's Hoare triple is divided up into separate Hoare triples for both the above cases during the proof of the top-level bootstrapping function (boot_sys).

Containment

For this statement, we prove that for each node, p_region_of_kobj_ut is contained in the region ndks_non_sh_p_reg. It is a more or less straight-forward Hoare-triple proof up the function call tree to bootstrapping part 2, where we take a node-local view. In init_node_state, the main function of part 2, the ghost field ndks_non_sh_p_reg is written. Hence, we prove that we initialise it with the non-shared memory region derived from the parameters of init_node_state, and we prove that all callees preserve p_region_of_kobj_ut being contained in the non-shared region.

Disjunction

For Hoare triples of callees of boot_sys, the style of comparing two nodes' non-shared memory regions changes to proving that each node's ghost field ndks_non_sh_p_reg is initialised correctly. Hence, these Hoare triples prove that after a node is initialised, its ndks_non_sh_p_reg field is initialised with the union of the memory regions covered by the node's user-level image and by the available memory region assigned to the node. This results in proof goals requiring us to show that all nodes' assigned available memory regions are chosen such that they do not overlap with (1) each other and (2) with any user-level image. We also need Axioms 5.4.6–5.4.10, which provide guarantees about where the boot loader places the user-level images. For example, we need the fact that no two nodes' user-level images overlap and that no user-level image overlaps with any node's available memory region.

■

Discussion

The kernel isolation theorem's Hoare-triple proofs over bootstrapping part 1 required 400 LOC whereas 730 LOC were necessary for part 2. This seems counter-intuitive at first sight, because the complexity of proving disjunction of the involved memory regions is in part 1. In part 2, we only have to handle containment. Hence, we would have expected the proof over part 2 to be larger than over part 1.

The reason for the increased complexity in part 2 is the detailed abstract specification. As mentioned before, the specification is particularly detailed with regards to dynamic memory allocation for object creation. This is exactly the kind of detail that matters for the kernel isolation theorem. Examples of details we model are:

- When accessing the kernel heap, the seL4 kernel does not directly access physical memory. Instead, memory is accessed through the kernel window, which is a region of virtual memory that maps to the physical memory used by the kernel heap. The mapping is linear with a constant offset. During bootstrapping, there are multiple occasions where physical addresses have to be converted into kernel-window addresses, and vice versa. We explicitly model these conversions, with the benefit that we have in fact discovered a conversion bug during the proof. It was only possible to complete the proof after this bug had been fixed.

- Bootstrapping-specific code and data are not needed during the runtime phase. In order to save memory, seL4::CMK reuses this *boot memory* after bootstrapping has finished. The abstract specification models this explicitly, which results in increased complexity of the non-shared memory region's definition and therefore proofs about it.

While the main purpose of the kernel isolation theorem is enabling a refinement proof by allowing us to reason about each node in isolation, the isolation statement of the theorem also has a benefit of its own. When designing systems on top of a clustered multi-kernel, we can place components that should be isolated into different nodes. The kernel isolation theorem, together with the user-level isolation theorem, guarantee isolation. In contrast, proving isolation between components within the same node is a research topic on its own [ABK12].

5.12.4 User-Level Theorems

This section presents the proofs of the user-level theorems, which we introduced in Section 5.8.

Theorem (User-Level Isolation)

```
⟦node_1_id < num_nodes; node_2_id < num_nodes⟧
⟹ {|λs. s = init_glms|} boot_sys
    {|λ_ s. ∃reg_1 reg_2.
              ndks_prop (λndks. reg_1 = ndks_non_sh_p_reg ndks) node_1_id s ∧
              ndks_prop (λndks. reg_2 = ndks_non_sh_p_reg ndks) node_2_id s ∧
              ndks_prop (λndks. p_region_of_non_sh_frame_caps ndks ⊆ reg_1)
               node_1_id s ∧
              ndks_prop (λndks. p_region_of_non_sh_frame_caps ndks ⊆ reg_2)
               node_2_id s ∧
              (node_1_id ≠ node_2_id ⟶ reg_1 ∩ reg_2 = ∅)|}
```

The structure of the proof is identical to the kernel isolation theorem. The only difference is that we prove containment of p_region_of_non_sh_frame_caps instead of p_region_of_kobj_ut.

■

Theorem (User-Level Sharing)

```
{|λs. s = init_glms|} boot_sys
{|λ_ s. ∃reg. ∀node_id<num_nodes.
                ndks_prop (λndks. p_region_of_sh_frame_caps ndks = reg)
                 node_id s|}
```

With regards to part 1 of bootstrapping, this is a straight-forward Hoare triple proof over all functions, with the *reg* region being instantiated in boot_sys, where the shared region is chosen.

The complexity lies in part 2 of bootstrapping, because this is where the shared frames are created. Furthermore, this theorem talks about equality of memory regions. This is much harder to prove than containment, such as in the previous two proofs. For example, in the loop where the shared frames are created, each iteration creates one frame that covers a part of the shared region. Therefore, we not only had to prove that each frame's covered memory region lies within the shared region, we also had to prove that after the loop has terminated, we have created all frames necessary to cover the entire shared region.

■

Discussion

The distribution of complexity mentioned above can also be seen in the LOC that were required to prove the user-level theorems. Bootstrapping part 1 required 470 LOC while part 2 required 1270 LOC.

5.12.5 Summary

Table 5.5 summarises the LOC counts for the proofs presented in Section 5.12.

Most of these proofs are Hoare-triple proofs. We can see that the LOC sum is more or less the same for both bootstrapping parts. Nevertheless, depending on the theorem, there are huge differences in where the complexity lies. We can also observe that there is a considerable amount of general proof effort that is applicable to all theorems presented here, and most likely to further theorems as well.

Proof Size	General	K. Mem. Seq. Acc.	Kernel Iso.	User Level	Total
Bootstr. Part 1	1230 LOC	2080 LOC	400 LOC	470 LOC	4180 LOC
Bootstr. Part 2	1330 LOC	360 LOC	730 LOC	1270 LOC	3690 LOC
Total	2560 LOC	2440 LOC	1130 LOC	1740 LOC	7870 LOC

Table 5.5: *Size of Bootstrapping Hoare-Triple Proofs*

5.13 Conclusion

In this chapter, we presented the part of the refinement lifting framework that is concerned with the bootstrapping phase of the kernel.

In Section 5.2, we introduced our TSO *multiprocessor execution model*, which augments the existing verification framework with support for concurrency. This model non-deterministically and symbolically runs an abstract parallel program and computes all possible resulting memory-access scenarios.

The state monads used in the specifications of seL4 and seL4::CMK assume that during the monadic execution, we observe sequential semantics. As a first step towards proving this assumption, we contributed the kernel-agnostic *sequential-semantics theorem* (Section 5.3), which uses the TSO model to prove that in a bootstrapping situation, every CPU observes sequential semantics, under the assumption that (1) memory fences are used correctly and (2) the disjoint-read/write-region property is adhered to. These assumptions have to be proved for the specification in question when applying this theorem.

Section 5.4 was devoted to formally specifying the code of a clustered multikernel's bootstrapping phase. First, we presented our approach to connecting a bootstrapping specification to the system boot loader. Second, we showed how a specification dynamically allocates and accesses data structures. Third, we explained how we connect a specification to the multiprocessor execution model. We did all of this with the example of an abstract specification of seL4::CMK's bootstrapping, which we provided as well.

In Section 5.5, we combined this specification with the sequential-semantics theorem. The result was the *kernel-memory-sequential-access theorem*, which states that during bootstrapping of seL4::CMK, each CPU observes sequential semantics. This was the assumption we made by using a non-deterministic state monad for seL4::CMK's abstract specification. We therefore proved that this assumption is correct.

At this point, Section 5.6 took a break with an intermediate conclusion. So far, he had concentrated on correctness of the bootstrapping phase itself. We continued with theorems that—while still talking about bootstrapping—are required by the runtime phase of the kernel.

An important design aspect of the clustered multikernel is that a node's kernel is isolated from other nodes. To this end, Section 5.7 introduced the *kernel isolation theorem*, which states that the nodes' kernels are bootstrapped isolated.

In Section 5.8, we presented the user-level theorems. They assert useful properties about user-level memory, which a user-level application can rely on. For example, together with the kernel isolation theorem, the user-level isolation theorem provides node-isolation guarantees which can be useful for building systems on top of seL4::CMK. Nevertheless, user-level theorems are not strictly necessary for refinement lifting.

In Section 5.9, we introduced a technique to connect seL4::CMK's abstract bootstrapping specification with the runtime-phase specification of seL4 (from the L4.verified project). This was necessary because the two specifications are based on slightly different formalisations.

Finally, in Section 5.10, we presented the multikernel refinement theorem. This theorem is applicable to a multikernel-version of seL4, i.e. a configuration of seL4::CMK where each node runs on exactly one CPU. The step to a *clustered multikernel*—i.e. multiple CPUs per node—is the topic of the next chapter (Chapter 6).

In Section 5.11, we discussed the assumptions we make about the compiler, boot loader and how we overapproximate memory accesses.

Finally, in Section 5.12, we presented the proofs of the theorems mentioned above and discussed their complexity.

Chapter 6

Refinement Lifting Framework: Runtime Phase

In this chapter, we present the part of the refinement lifting framework that is concerned with the runtime phase of the kernel. For this phase, the kernel isolation theorem (Section 5.7) allows us to reason about each node in isolation, which makes verification more tractable. Consequently, this chapter takes a node-local view.

Recall from Section 5.9 that during the runtime phase, a clustered-multikernel node of size 1 (i.e. running on one CPU) is indistinguishable (on the abstract level) from a uniprocessor kernel. This reduces the problem in the sense that we can assume we have a valid model and refinement proof for a node of size 1. Lifting the runtime phase of the kernel is therefore reduced to lifting a node of size 1 to a node of size n.

Consequently, lifting the runtime phase consists of: (1) adapting the model of a node to support multiple CPUs running in parallel instead of one, (2) adapting the refinement statement to the new model, and (3) fixing up the refinement proof.

Approach There are two fundamentally different ways to do this: We can either (1) directly modify the Isabelle code of the original model, refinement statement and proofs, or (2) leave the original model/proofs untouched and write new Isabelle code which adapts the original model/proofs accordingly. While the first option offers maximum flexibility in the adaption process, the second option enables a more generic handling of the problem, independent of the kernel, which aids reusability. Furthermore, the separation of the refinement lifting problem from kernel specifics enhances readability of the model and decouples proofs. For these reasons, we focus on the second option, despite its reduced flexibility regarding the adaption process.

As we recall from Section 2.6.3, in seL4's refinement proof, the ADTs define a *global automaton*, which models the actions of the kernel on a single CPU. Within a node of a clustered multikernel, we have multiple CPUs running in parallel. This suggests modelling a node as a parallel composition of the original automaton. Nevertheless, a simple parallel composition of states and interleaving of steps is not a valid model. From Section 4.5 we know that the kernel state within a node consists of the original uniprocessor state, split up into two parts: (1) local (to a CPU) and (2) shared (between CPUs of that node). The shared part inherits most of the uniprocessor state, e.g. the kernel heap (containing kernel objects). The local part is duplicated for each CPU. It consists of the abstract CPU state (registers, kernel/user/idle mode) and most importantly, the pointer to the currently running thread.

Challenges We need to take into account the state splitting and hence the possibility that whenever one of the parallel automata makes a step, it potentially modifies its own local state and the shared state (while it is not allowed to modify another CPU's state). As such, it is possible that from another CPU's point of view, the shared state gets changed magically under its feet while its local state remains unchanged; something that cannot happen to the state in the original model. We also need to address the problem that in most cases, the interleaved steps are bigger than what is executed atomically on the machine level.

6.1 Chapter Overview

This chapter is organised as follows: Section 6.2 introduces the lifting operation and theorems, which are the formal foundation of runtime-phase refinement lifting. To show their practicability, we apply them to the seL4 microkernel in Section 6.3. However, a successful application requires solving the *running-thread problem* (Section 6.4). In Section 6.5, we present the resulting top-level *clustered-multikernel refinement theorem*. In Section 6.6, we discuss limitations and the assumptions we make for the runtime-phase refinement lifting, and conclude in Section 6.7.

The basic ideas behind runtime-phase refinement lifting have already been published in our previous work [vT12].

6.2 Lifting Operation/Theorems

This thesis contributes a formal *lifting operation* with accompanying *refinement lifting theorem* and *invariant lifting theorem*. The job of the lifting operation is to turn the original model into a valid model of a node running on multiple CPUs. It does this by lifting the original automaton into a parallel composition of itself with a parameterisable splitting of the original state into shared and local parts ("loc" in Figure 6.1). Steps of multiple CPUs are interleaved non-deterministically. The lifting operation also defines what refinement means in this context.

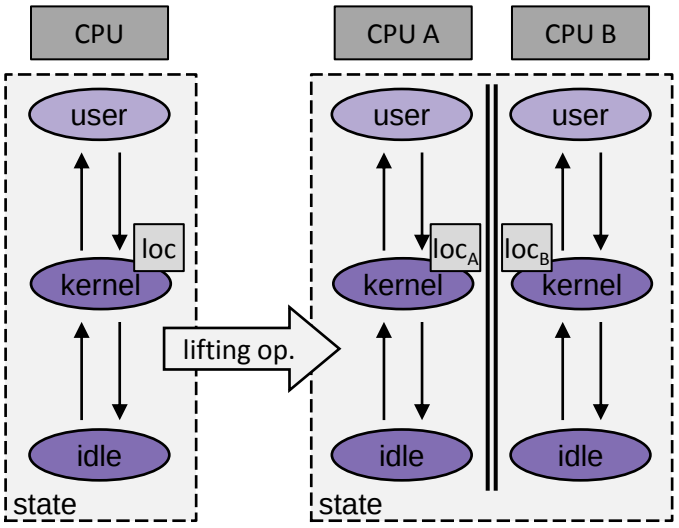

Figure 6.1: *Lifting into Parallel Composition*

The *refinement lifting theorem* proves this refinement under the assumption that the

original refinement holds and was proved via forward simulation. More precisely, it states: "When applying the lifting operation to the ADTs of both abstract and concrete levels of a forward-simulation proof, the concrete parallel ADT forward-simulates the abstract parallel ADT if the concrete original ADT forward-simulates the abstract original ADT."

If the original forward-simulation proof relies on invariants, the *invariant lifting theorem* lifts the original invariants into the context of the parallel ADT.

The lifting operation and theorems are kernel-agnostic, i.e. they are independent from seL4's code and proof base and can also be applied to kernels other than seL4 under the following condition: Refinement of the kernel in question must have been proved via forward simulation with the calculus presented in Section 2.2.

The lifting operation makes certain assumptions about interleaving and atomicity of abstract steps which will be discussed in Section 6.6. The remainder of this section formally defines the lifting operation and theorems on top of the refinement calculus presented in Section 2.2.

6.2.1 Lifting Operation

The lifted, parallel ADT's observable and private state types are defined as follows.

Definition 6.2.1 (Lifted Observable and Private State Types)

```
types 'obs obs_t          = cpu_id_t ⇒ 'obs
types ('loc, 'sh) priv_t  = (cpu_id_t ⇒ 'loc) × 'sh
```

The observable state is the aggregation of what every CPU observes, i.e. a function from a CPU ID to an original observable state. The private state consists of parameterisable shared and local parts. The local part exists for each CPU.

The lifting operation consists of two *lifting functions*, which will appear in the refinement lifting theorem's assumptions. The first one is the ADT lifting function. Lifting of an ADT is defined by lifting each of its three functions.

Definition 6.2.2 (ADT Lifting)

```
lift_adt :: ('priv, 'obs, 'j) adt_t
          ⇒ ('loc ⇒ 'sh ⇒ 'priv)
             ⇒ (('loc, 'sh) priv_t ⇒ ('loc, 'sh) priv_t ⇒ bool)
                ⇒ (('loc, 'sh) priv_t, 'obs obs_t, cpu_id_t × 'j) adt_t
lift_adt A wrap init_upd ≡ (|Init = ⋃init_upd ` lift_init (Init A) wrap,
                             Fin = lift_fin (Fin A) wrap,
                             Step = lift_step (Step A) wrap|)
```

The parameter `wrap :: 'loc ⇒ 'sh ⇒ 'priv` is the *wrapping function*, which defines how the original state is split up into shared and local parts. It does this by wrapping a given shared and a given local part into an original state. It is passed to all three lifting subfunctions. Examples of wrapping functions will be given in Definition 6.3.2.

Initialisation can be further parameterised with the function `init_upd`, which provides the possibility to update the private states of the parallel ADT after bootstrapping in order to account for different local initial states.

Lifting of Init is defined as follows.

Definition 6.2.3 (Init Lifting)

```
lift_init :: ('priv ⇒ bool) ⇒ ('loc ⇒ 'sh ⇒ 'priv) ⇒ ('loc, 'sh) priv_t ⇒ bool
lift_init init wrap ≡ {(floc, sh) | ∃loc. ∀c. floc c = loc ∧ wrap loc sh ∈ init}
```

Each initial state returned by the original Init function is split up into its local and shared part. The local part is duplicated for each CPU. The function `floc :: cpu_id_t ⇒ 'loc` holds all local parts of the state.

This definition of Init lifting results in all local states being identical. While this correctly models that all CPUs have an identical state after bootstrapping, it does not capture things like different CPUs running different initial threads. For example, after a clustered multikernel is bootstrapped, the first CPU runs the initial thread of its node while the remaining CPUs run the idle thread. We will address this problem with the parameterisation function `init_upd`.

Lifting of Step is defined as follows.

Definition 6.2.4 (Step Lifting)

```
lift_step :: ('j ⇒ 'priv × 'priv ⇒ bool)
          ⇒ ('loc ⇒ 'sh ⇒ 'priv)
             ⇒ cpu_id_t × 'j
                ⇒ ('loc, 'sh) priv_t × ('loc, 'sh) priv_t ⇒ bool
lift_step step wrap ≡
    λ(c, j). {(((floc, sh), floc', sh') | (wrap (floc c) sh, wrap
    (floc' c) sh') ∈ step j ∧ (∀x. x ≠ c ⟶ floc x = floc' x)}
```

We extend the original transition `j` with the ID `n` of the CPU on which this transition (and the resulting steps) takes place. Each step in the new parallel ADT consists of (1) wrapping the shared state and the current CPU's local state into the original state by using the provided wrapping function; (2) executing the original step and (3) splitting up the resulting state back into the new shared and local states. The local states of the other CPUs are not changed. This definition interleaves the different CPUs' steps non-deterministically. Note that this definition of interleaving makes certain assumptions about atomicity of steps which will be discussed in Section 6.6.

Lifting of Fin is defined as follows.

Definition 6.2.5 (Fin Lifting)

```
lift_fin :: ('priv ⇒ 'obs)
         ⇒ ('loc ⇒ 'sh ⇒ 'priv) ⇒ ('loc, 'sh) priv_t ⇒ 'obs obs_t
lift_fin fin wrap ≡ λpriv c. fin (wrap (fst priv c) (snd priv))
```

The lifted Fin function combines each CPU's local private state with a copy of the shared private state, wraps each pair and returns the resulting observable state as defined in Definition 6.2.1.

The second lifting function lifts the refinement relation.

Definition 6.2.6 (Refinement-Relation Lifting)

```
lift_rel :: ('priv_A × 'priv_C ⇒ bool)
         ⇒ ('loc_A ⇒ 'sh_A ⇒ 'priv_A)
            ⇒ ('loc_C ⇒ 'sh_C ⇒ 'priv_C)
               ⇒ ('loc_A, 'sh_A) priv_t × ('loc_C, 'sh_C) priv_t ⇒ bool
lift_rel R wrap_A wrap_C ≡ {(((floc_A, sh_A), floc_C, sh_C) | ∀c. (wrap_A
                           (floc_A c) sh_A, wrap_C (floc_C c) sh_C) ∈ R}
```

Two states in the lifted ADT are related if, for all CPUs, their wrapped counterparts are related in the original ADT.

6.2.2 Refinement Lifting Theorem

Based on the lifting functions above, we formulate the refinement lifting theorem as follows.

Theorem 6.2.7 (Refinement Lifting)

$$\frac{\begin{array}{c}\text{fw\_sim\_inv } A\ C\ R\ I \\ A' = \text{lift\_adt } A\ wrap\_A\ init\_upd\_A \qquad C' = \text{lift\_adt } C\ wrap\_C\ init\_upd\_C \\ R' = \text{lift\_rel } R\ wrap\_A\ wrap\_C \qquad I' = \text{lift\_rel } I\ wrap\_A\ wrap\_C \qquad \text{wrap\_surjective } wrap\_A \\ \text{rel\_splittable } R\ wrap\_A\ wrap\_C \qquad \text{init\_upd\_corres } R'\ init\_upd\_A\ init\_upd\_C\end{array}}{\text{fw\_sim\_inv } A'\ C'\ R'\ I'}$$

The goal states that the parallel ADT C' forward-simulates the parallel ADT A' under refinement relation R' with invariants I'. It pairs up with the first assumption, which states that the original ADT C forward-simulates the original ADT A under refinement relation R with invariants I.

The next four assumptions use the lifting functions to define the parallel ADTs based on the original ADTs, using the parameterisation functions *wrap_A*, *wrap_C* and *init_upd*.

The last three assumptions restrict the parameterisation space, since the parameterisation needs to meet certain conditions in order for the refinement lifting theorem to be true. They are explained in the following three definitions.

Definition 6.2.8 (Surjective Wrapping Function)

```
wrap_surjective :: ('loc ⇒ 'sh ⇒ 'priv) ⇒ bool
wrap_surjective wrap ≡ ∀priv. ∃loc sh. priv = wrap loc sh
```

This assumption arises because a non-surjective wrapping function means we lose information when splitting an original private state. Consequently, splitting and then wrapping a set of such states results in a subset of the original states. It is interesting that only the abstract wrapping function needs to be surjective, but not the concrete one. The explanation is that getting back a subset of the original states on the concrete level does not violate refinement because the subset is still the subset of the related abstract states. Nevertheless, a parameterisation with a non-surjective concrete wrapping function is most likely flawed, i.e. the resulting refinement statement is most likely not what the author intended it to be.

Definition 6.2.9 (Splittable Refinement Relation)

```
rel_splittable :: ('priv_A × 'priv_C ⇒ bool)
                  ⇒ ('loc_A ⇒ 'sh_A ⇒ 'priv_A)
                     ⇒ ('loc_C ⇒ 'sh_C ⇒ 'priv_C) ⇒ bool
rel_splittable R wrap_A wrap_C ≡
    ∃L S. ∀loc_A sh_A loc_C sh_C. ((wrap_A loc_A sh_A, wrap_C
    loc_C sh_C) ∈ R) = (L loc_A loc_C ∧ S sh_A sh_C)
```

This function determines if the refinement relation R is *splittable* with regards to the wrapping functions *wrap_A* and *wrap_C*. Splittable means that it does not relate states that are in the shared part on the abstract level but in the local part on the concrete level (or vice versa).

Definition 6.2.10 (Init-Update Correspondence)

```
init_upd_corres :: (('loc_A, 'sh_A) priv_t × ('loc_C, 'sh_C) priv_t ⇒ bool)
                   ⇒ (('loc_A, 'sh_A) priv_t ⇒ ('loc_A, 'sh_A) priv_t ⇒ bool)
                      ⇒ (('loc_C, 'sh_C) priv_t ⇒ ('loc_C, 'sh_C) priv_t ⇒ bool)
                         ⇒ bool
init_upd_corres R' init_upd_A init_upd_C ≡
    ∀s_a s_c. (s_a, s_c) ∈ R' ⟶ init_upd_C s_c ⊆ R' `` init_upd_A s_a
```

This function determines if the abstract and concrete `init_upd` functions correspond.

6.2.3 Invariant Lifting Theorem

Recall from Section 2.2 that the property fw_sim_inv in the goal of the refinement lifting theorem means that forward simulation holds *under the assumption* that the invariants hold. Therefore, we need a theorem that lifts the existing original invariant proofs.

Definition 6.2.11 (Invariant Lifting)

```
lift_invs :: ('priv ⇒ bool) ⇒ ('loc ⇒ 'sh ⇒ 'priv) ⇒ ('loc, 'sh) priv_t ⇒ bool
lift_invs I wrap ≡ {(floc, sh) | ∀c. wrap (floc c) sh ∈ I}
```

In the lifted ADT, a state satisfies the lifted invariants if, for all CPUs, its wrapped counterpart satisfies the original invariants.

Theorem 6.2.12 (Invariant Lifting)

$$\frac{A \models I \qquad A' = \text{lift\_adt}\ A\ wrap\ init\_upd \qquad I' = \text{lift\_invs}\ I\ wrap \qquad \text{invs\_splittable}\ I\ wrap \qquad \bigcup init\_upd\ `\ I' \subseteq I'}{A' \models I'}$$

The goal pairs up with the first assumption: The lifted invariants I' hold in ADT A' if the original invariants I hold in ADT A. The second and third assumption define the lifting whereas the last two assumptions are conditions on the parameterisation. The last one requires the `init_upd` function to preserve the lifted invariants I' while the second last one requires the original invariants I to be *splittable* with regards to the wrapping function `wrap`.

Definition 6.2.13 (Splittable Invariants)

```
invs_splittable :: ('priv ⇒ bool) ⇒ ('loc ⇒ 'sh ⇒ 'priv) ⇒ bool
invs_splittable I wrap ≡
    ∃L S. ∀sh floc c. (wrap (floc c) sh ∈ I) = (L (floc c) ∧ S sh)
```

The property states that the invariants do not relate shared and local parts of the state to each other in any way.

Using the goals of the refinement lifting theorem and the invariant lifting theorem as the assumptions of Theorem 2.2.9 shows that forward simulation holds for the parallel (lifted) ADT, which also implies refinement according to Theorem 2.2.6.

We would now be ready to apply the lifting operation and theorems to seL4's uniprocessor proof. Unfortunately, this is not yet possible because, as Section 6.3 will show, the splittable invariants property does not hold in seL4. Specifically, a few invariants talk about the state of the currently running thread. These invariants are not splittable because the pointer to the currently running thread is in the local part of the node state while all thread states are in the shared part of the node state. Therefore, we need a way to deal with unsplittable invariants separately from splittable ones. To this end, we generalise the invariant lifting theorem in the following way.

Theorem 6.2.14 (Invariant Lifting with Separation)

$$\frac{A \models I_s \cap I_u \qquad A' = \text{lift\_adt}\ A\ wrap\ init\_upd \qquad I_s' = \text{lift\_invs}\ I_s\ wrap \qquad I_u' = \text{lift\_invs}\ I_u\ wrap \qquad \text{invs\_splittable}\ I_s\ wrap \qquad \bigcup init\_upd\ `\ I_s' \subseteq I_s' \qquad \text{Init}\ A' \subseteq I_u' \qquad \forall c\ j.\ \text{Step}\ A'\ (c, j)\ ``\ (I_s' \cap I_u') \subseteq I_u'}{A' \models I_s' \cap I_u'}$$

It is generalised in the way that instantiating tautological unsplittable invariants (I_u = UNIV and I_u' = UNIV) and simplifying yields the invariant lifting theorem presented before.

The new theorem allows separating splittable invariants I_s and unsplittable invariants I_u. Splittable invariants are lifted automatically while unsplittable ones have to be proved manually in the context of the parallel (lifted) ADT.

The last two assumptions are new. The first of them requires that the unsplittable invariants are established by Init, the second requires that they are preserved by the Step function. The reader might wonder why we do not use the term $A' \models I_u'$ instead of the two separate assumptions. The reason is a small but important detail: Unfolding that term does yield the two assumptions, except that I_s' is missing. The problem is that without it being there, we would not be able to use the splittable invariants in order to prove the unsplittable ones, which is vital.

Moreover, as we will see in Section 6.4, proving the unsplittable invariants for seL4 requires an additional invariant in the context of the parallel ADT. To accommodate for this, we further generalise the invariant lifting theorem with separation.

Theorem 6.2.15 (Invariant Lifting with Separation and Additional Invariants)

$$\frac{\begin{array}{c} A \models I_s \cap I_u \qquad A' = \text{lift\_adt}\ A\ wrap\ init\_upd \qquad I_s' = \text{lift\_invs}\ I_s\ wrap \\ I_u' = \text{lift\_invs}\ I_u\ wrap \qquad \text{invs\_splittable}\ I_s\ wrap \qquad \bigcup init\_upd\ `\ I_s' \subseteq I_s' \\ \text{Init}\ A' \subseteq I_u' \cap I_a' \qquad \forall c\ j.\ \text{Step}\ A'\ (c, j)\ ``\ (I_s' \cap I_u' \cap I_a') \subseteq I_u' \cap I_a' \end{array}}{A' \models I_s' \cap I_u' \cap I_a'}$$

It is identical to the invariant lifting theorem with separation, except for the addition of the invariants I_a' in the goal and the last two assumptions. The additional invariants aid the proof of the unsplittable invariants which, otherwise, would be very hard or impossible to prove. An example based on seL4 will be discussed in Section 6.4.

Note that we presented the three invariant lifting theorems in this sequence for educational purposes. In Isabelle, only the most generalised version (Theorem 6.2.15) is directly proved and applied to seL4. The other two are derived by instantiation and automatic simplifying.

The specification of the lifting operation in Isabelle comprises 150 LOC. The proofs of the lifting theorems comprise 120 LOC.

6.2.4 Assumptions

The lifting operation interleaves the steps of the original automaton non-deterministically. It thereby assumes that interleaving semantics are applicable. This assumption was also made in Back's refinement of parallel programs [Bac90]. As he states, modelling a parallel system as a sequential non-deterministic interleaving of its steps is only correct if (1) the steps are atomic, or (2) they do not interfere (with regards to the state they access).

This means that applying the lifting operation to a kernel only models reality correctly if all combinations of that kernel's transitions either are atomic or do not interfere with each other. We perform such an analysis with the example of seL4 in Section 6.6.

6.2.5 Summary

We contributed a formal lifting operation with accompanying refinement lifting theorem and invariant lifting theorem. The job of the lifting operation is to turn the original model into a valid model of a node running on multiple CPUs. It does this by lifting the original automaton into a parallel composition of itself with a parameterisable splitting of the

original state into shared and local parts. Steps of multiple CPUs are interleaved non-deterministically. The lifting operation also defines what refinement means in this context. We assume that interleaving semantics are applicable.

In case there are unsplittable invariants, they can be separated and handled manually. While this makes handling them kernel-specific, experience with seL4 (Section 6.3) shows that very few invariants are unsplittable. Furthermore, they are already proved for the original ADT. The additional proof only has to manually lift them into the parallel context.

The lifting operation and theorems presented in this thesis are—to the best of our knowledge—the first to automatically lift an automaton of a refinement proof into a parallel composition of itself with parameterisable splitting of the original state into shared and local parts.

6.3 Application to seL4

In this section, we present how we formally apply the lifting operation and theorems to seL4. We start with the *original ADT* from Section 5.10, which models a node running on one CPU.

6.3.1 Instantiation of Type Variables

First, we have to define how the original state is split into a local and a shared part. The following definitions are based on the original seL4 types introduced in Section 2.6.3.

The types of the abstract and concrete[1] parallel ADTs' local and shared states are defined as follows.

Definition 6.3.1

types *cmk_loc_A_t* = *(user_context × word32) × mode × event option*
types *cmk_loc_C_t* = *(user_context × word32) × mode × event option*
types *cmk_sh_A_t* = *state*
types *cmk_sh_C_t* = *kernel_state*

In the abstract and concrete local states, the type *user_context* contains the CPU's register set, *mode* defines whether we are in kernel, user or idle mode, and *event option* captures whether an event just happened, e.g. an IRQ was triggered. Most importantly, the local state contains the pointer to the currently running thread as *word32*. Note that it is a coincidence that the local states of both abstract and concrete levels have the same type. It is not necessary for the lifting operation and theorems to be applicable. The reason is that the four subparts of the local state happen to be modelled at the same level of detail (using the same types) in the original seL4 model.

The abstract and concrete shared states consist of the respective original kernel monad state record, minus the pointer to the currently running thread. Instead of defining a new record containing all fields except that one, we conveniently use the respective original type and ignore this field.

6.3.2 Parameterisation Functions

With the lifting operation's type variables instantiated, we are ready to define the first parameterisation function *wrap*.

[1] Note that what we call *concrete* ADT/level/specification in this section refers to seL4's intermediate ADT/level/specification.

Definition 6.3.2 (Abstract and Concrete Wrapping Functions)

```
cmk_wrap_A :: cmk_loc_A_t ⇒ cmk_sh_A_t ⇒ state global_state
cmk_wrap_A loc sh ≡
    let ((uc, ct), m, eo) = loc in ((uc, sh(|cur_thread := ct|)), m, eo)

cmk_wrap_C :: cmk_loc_C_t ⇒ cmk_sh_C_t ⇒ kernel_state global_state
cmk_wrap_C loc sh ≡
    let ((uc, ct), m, eo) = loc in ((uc, sh(|ksCurThread := ct|)), m, eo)
```

Using the same underlying record type for the shared state `cmk_sh_A_t` and the original state `state global_state` makes the wrapping functions less verbose because we can just copy the shared state into the original state and update the current-thread pointer.

Before we can define the lifting operation, we need to define the second parameterisation function `init_upd`.

Definition 6.3.3 (Abstract and Concrete Init-Update Functions)

```
init_upd_A :: (cmk_loc_A_t, cmk_sh_A_t) priv_t ⇒
              (cmk_loc_A_t, cmk_sh_A_t) priv_t ⇒ bool
init_upd_A (floc, sh) ≡
    {(floc', sh') | sh' = sh ∧ floc' 0 = floc 0 ∧ (∀c>0. floc' c =
    ((empty_context, idle_thread sh), IdleMode, None))}

init_upd_C :: (cmk_loc_C_t, cmk_sh_C_t) priv_t ⇒
              (cmk_loc_C_t, cmk_sh_C_t) priv_t ⇒ bool
init_upd_C (floc, sh) ≡
    {(floc', sh') | sh' = sh ∧ floc' 0 = floc 0 ∧ (∀c>0. floc' c =
    ((empty_context, ksIdleThread sh), IdleMode, None))}
```

These functions leave untouched the shared state and the locate state of the first CPU, which is already correctly initialised to run the initial thread of the node. The other local states, however, need to be updated and configured to run the idle thread. This is done by updating their current-thread pointers to point to the idle thread and by setting the `mode` to IdleMode.

6.3.3 ADTs of seL4::CMK

Finally, we can define the lifted seL4::CMK ADTs.

Definition 6.3.4 (Lifted seL4::CMK ADTs)

```
cmk_ADT_A :: node_id_t ⇒ ((cmk_loc_A_t, cmk_sh_A_t) priv_t,
             observable obs_t, cpu_id_t × global_transition) adt_t
cmk_ADT_A node_id ≡ lift_adt (orig_ADT_A node_id) cmk_wrap_A init_upd_A

cmk_ADT_C :: node_id_t ⇒ ((cmk_loc_C_t, cmk_sh_C_t) priv_t,
             observable obs_t, cpu_id_t × global_transition) adt_t
cmk_ADT_C node_id ≡ lift_adt (orig_ADT_C node_id) cmk_wrap_C init_upd_C
```

Remember that the original ADTs orig_ADT_A and orig_ADT_C are defined in Section 5.10 and represent a node running on one CPU.

6.3.4 Proofs

Before we can apply the refinement lifting theorem, we need to prove the following three lemmas.[2]

Lemma 6.3.5 *The abstract wrapping function is surjective.*

```
wrap_surjective cmk_wrap_A
```

Lemma 6.3.6 *The original refinement relation is splittable.*

```
rel_splittable refine_rel cmk_wrap_A cmk_wrap_C
```

Lemma 6.3.7 *The abstract and concrete* `init_upd` *functions correspond.*

```
init_upd_corres (lift_rel refine_rel cmk_wrap_A cmk_wrap_C) init_upd_A init_upd_C
```

Now we are ready to apply the refinement lifting theorem (Theorem 6.2.7) as we have defined/proved all of its assumptions. Note that the first assumption (forward simulation of the original ADTs) was one of the key lemmas we proved in Section 5.10 (Lemma 5.10.4). The application of the refinement lifting theorem results in the following lemma.

Lemma 6.3.8 *Forward simulation holds for the parallel ADTs* cmk_ADT_A *and* cmk_ADT_C *(under the assumption that the lifted invariants hold).*

```
node_id < num_nodes ⟹
fw_sim_inv (cmk_ADT_A node_id) (cmk_ADT_C node_id)
  (lift_rel refine_rel cmk_wrap_A cmk_wrap_C)
  (lift_invs full_invs cmk_wrap_A × lift_invs full_invs' cmk_wrap_C)
```

In order to prove forward simulation (and therefore refinement, our final goal), we have to prove that the lifted invariants hold. This could be proved by using the invariant lifting theorem (Theorem 6.2.12). Unfortunately, as mentioned earlier, not all of seL4's invariants are splittable. This means that we have to use the invariant lifting theorem with separation of unsplittable invariants (Theorem 6.2.14). We start with defining which of seL4's invariants are unsplittable.

Definition 6.3.9 (Unsplittable Invariants)

```
orig_invs_unsplit :: state global_state ⇒ bool
orig_invs_unsplit ≡ {((uc, s), m,
                      e) | (ct_running s ∨ ct_idle s) ∧
                           (m = UserMode ⟶ ct_running s) ∧
                           (m = IdleMode ⟶ ct_idle s) ∧
                           ((∃y. e = ⌊y⌋) ∧ e ≠ ⌊Interrupt⌋ ⟶ ct_running s)}

orig_invs'_unsplit :: kernel_state global_state ⇒ bool
orig_invs'_unsplit ≡ {((uc, s), m,
                       e) | sch_act_wf (ksSchedulerAction s) s ∧
                            (ct_running' s ∨ ct_idle' s) ∧
                            (m = UserMode ⟶ ct_running' s) ∧
                            (m = IdleMode ⟶ ct_idle' s) ∧
                            ((∃y. e = ⌊y⌋) ∧ e ≠ ⌊Interrupt⌋ ⟶ ct_running' s)}
```

The unsplittable invariants boil down to ct_running and ct_idle, which state that the current-thread pointer points to a thread which is in state Running or IdleThreadState, respectively. Naturally, the splittable invariants are defined such that the following lemma holds.

[2] Proving these lemmas required 60 LOC in Isabelle.

Lemma 6.3.10 *On both abstract and concrete levels, the original invariants* full_invs *are a conjunction of the splittable and unsplittable invariants.*

full_invs = orig_invs_split ∩ orig_invs_unsplit
full_invs′ = orig_invs′_split ∩ orig_invs′_unsplit

In order to satisfy the assumptions of the invariant lifting theorem with separation, we have to prove the following three lemmas.[3]

Lemma 6.3.11 *On both abstract and concrete levels, the original splittable invariants are splittable.*

invs_splittable orig_invs_split cmk_wrap_A
invs_splittable orig_invs′_split cmk_wrap_C

Lemma 6.3.12 *On both abstract and concrete levels, the* `init_upd` *functions preserve the lifted splittable invariants.*

⋃init_upd_A ` lift_invs orig_invs_split cmk_wrap_A ⊆ lift_invs orig_invs_split cmk_wrap_A
⋃init_upd_C ` lift_invs orig_invs′_split cmk_wrap_C ⊆ lift_invs orig_invs′_split cmk_wrap_C

Lemma 6.3.13 *On both abstract and concrete levels, the lifted initialisation function establishes the lifted unsplittable invariants.*

$$\frac{\texttt{node\_id} < \text{num\_nodes}}{\text{Init } (\text{cmk\_ADT\_A } \texttt{node\_id}) \subseteq \text{lift\_invs orig\_invs\_unsplit cmk\_wrap\_A}}$$

$$\frac{\texttt{node\_id} < \text{num\_nodes}}{\text{Init } (\text{cmk\_ADT\_C } \texttt{node\_id}) \subseteq \text{lift\_invs orig\_invs}'\text{\_unsplit cmk\_wrap\_C}}$$

6.3.5 Summary

We have formally applied the lifting operation and theorems to seL4. This involved instantiating the lifting operation's type variables, defining the parameterisation functions (`wrap` and `init_upd`), and identifying/separating seL4's unsplittable invariants.

We have proved (1) the required properties about the parameterisation functions, (2) that the splittable invariants are splittable, and (3) that the unsplittable invariants are established.

The only lemma that remains to be proved in order to be able to apply the invariant lifting theorem (and ultimately prove refinement) is that the lifted Step function preserves the lifted unsplittable invariants. Unfortunately, this lemma is not true in the model and implementation of seL4::CMK presented so far. The reason is the *running-thread problem*, which we explain and solve in the next section.

6.4 Running-Thread Problem

As Figure 6.2 shows, the seL4 API allows a thread A in possession of a capability to thread B to modify (e.g. suspend, resume) or delete that thread.[4] On a uniprocessor, the kernel assumes that thread B, when modified/deleted by thread A, is not currently running on a CPU. In seL4::CMK however, thread B could be running on another CPU.

[3] Proving these lemmas required 160 LOC in Isabelle.

[4] We use *thread A* and *thread B* as example threads throughout this section. They represent any two threads in the node.

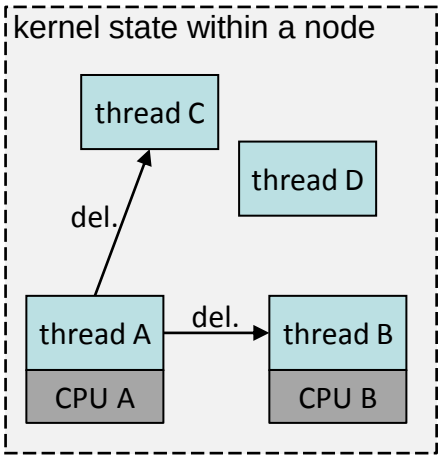

Figure 6.2: *Running-Thread Problem*

Its TCB cannot be directly modified/deleted without coordination as this would result in corruption as soon as thread B enters the kernel again. Formally, the unsplittable invariants are violated in such a case: When thread B enters the kernel, the ct_running property does not hold anymore because, for example, thread A has set thread B's state to Inactive while it was running in UserMode. Similarly, if thread B was deleted by thread A while it was running in UserMode, ct_running does not hold anymore because thread B's current-thread pointer does not point to a valid TCB anymore.

6.4.1 Solution

In order to solve this problem, we have to add coordination to seL4::CMK that prevents modification/deletion of threads that are currently running on another CPU. Before this problem is solved, the invariants do not hold and refinement cannot be proved (besides the fact that this problem is not desirable to have in an implementation anyway).

First, we need a way for thread A to detect whether thread B is currently running on another CPU. The TCB's field that stores the thread's current state would be a good candidate, but it does not offer this possibility because it is set to the state Running in both of the following cases: (1) the thread is currently running on a CPU, and (2) it is not, but it is in the ready queue. This is because it is not necessary to distinguish these two cases in the uniprocessor version. Consequently, we introduce a new thread state Current which is used for the first case, whereas Running is used for the second case.

Thread A can now detect if thread B is in state Current. This check has to be performed by thread A each time it tries to modify/delete thread B. If thread B is in state Current, thread A has to wait. However, waiting in the middle of an operation of an event-based kernel is impossible to implement. Furthermore, the big-lock model would not be applicable anymore: Thread B cannot enter the kernel (and have its state set to Running) as long as thread A still has the lock (while waiting for thread B), which would result in a deadlock.

Therefore, we solve the problem in an event-based style. Recall from Section 2.5.7 that seL4 has explicit preemption points in long-running operations. Between each iteration of these operations, seL4 checks for pending IRQs, and if detected, triggers a preemption exception. The exception is propagated up to the main kernel entry function where we leave the kernel and return to UserMode. Only here, the IRQ is triggered and we enter the kernel again to handle the IRQ. After having handled the IRQ, the kernel returns to UserMode, where the original system call is restarted and the long-running operation

continued.

Consequently, we use the preemption-exception mechanism to preempt any operation that is about to modify/delete another thread which is in state Current. The exception means we leave the kernel, and back in UserMode, the original system call is restarted. We enter the kernel again and execute the same system call, up to the same check. This KernelMode/UserMode busy-wait loop is repeated until thread B has entered the kernel and its state is set to Running. Thread B is able to enter the kernel because thread A periodically releases the big lock.

This solution also handles the case where multiple threads race for modifying one specific thread.

The only remaining problem is starvation, i.e. it could be possible that due to unlucky timing, thread A never gets a chance to modify thread B. We solve this problem by forcing thread B to reach a quiescent state eventually when thread A is waiting for it. To this end, we introduce another thread state: CurrentReqInactive. This state serves as a flag. It is semantically equivalent to and only reachable from state Current. Thread A sets this state in thread B's TCB to signal that it is waiting to perform an operation on it. Thread B is suspended (set to Inactive) the next time it enters the kernel to make it available for modification/deletion.

Note that this solution does not need IPIs (inter-processor interrupts) in order to work correctly. Nevertheless, using them might be beneficial for performance reasons because they reduce the time thread A has to wait for thread B. Also note that this solution does not touch code written to turn seL4 into seL4::CMK. Hence, the solution can be implemented in the uniprocessor version already. It has no impact on it, except for a few additional checks for thread states which always succeed in a uniprocessor environment. The checks are simple memory accesses and they occur only in rarely-called slow-path operations.

Even though the running-thread problem as presented here is specific to the seL4 API, it will arise in other kernels that allow manipulation of threads on other CPUs.

6.4.2 Formal Implications

Implementing the solution required only 100 LOC of C code and a similar amount on the abstract level. In total, two thread states and eight preemption points had to be added. Note that this is a new way of using preemption points in seL4. In the uniprocessor version, preemption points are conditional as well, but the condition is always a non-deterministic occurrence of a pending IRQ. For the preemption points we added here, the conditions are deterministic. Nevertheless, our experience is that the proof complexity they add is similar to non-deterministic preemption points.

Fixing up the invariant proofs over the modified abstract specification required us to modify 100 LOC of proof script and add 300 LOC. The main complexity comes from the added preemption points: We need to prove for each one that the invariants also hold if the preemption exception is taken.

Remember that the only lemma that remains to be proved in order to be able to apply the invariant lifting theorem is that the lifted Step function preserves the lifted unsplittable invariants. This property is now true in the new seL4::CMK version with the fixed running-thread problem. Its direct proof in Isabelle comprises 100 LOC.

However, this proof requires two additional invariants. First, we need to ensure that no thread in state Current or CurrentReqInactive is deleted/modified by a kernel call, unless it is the thread currently running on the CPU executing the kernel call.

Theorem 6.4.1 (Current-Threads Preservation)

```
{|λs. invs s ∧
      valid_duplicates s ∧
      (e ≠ Interrupt ⟶ ct_running s) ∧
      st_tcb_at current tcb s ∧ tcb ≠ cur_thread s|}
kernel_entry e uc
{|λ_ s. st_tcb_at current tcb s ∧ tcb ≠ cur_thread s|}
```

It is a Hoare triple over kernel_entry, which is the function that makes up the KernelTransition of the abstract original ADT, i.e. it is the function that contains the entire kernel functionality. The first three preconditions of the Hoare triple are simply a subset of the original invariants. The two last assumptions are the interesting ones: st_tcb_at current *tcb s* states that the thread *tcb* points to is in state current, whereas current is an abbreviation that comprises the states Current and CurrentReqInactive. The precondition *tcb* ≠ cur_thread *s* restricts the Hoare triple to only be applicable if the *tcb* in question is not the currently-running thread.

The postcondition consists of the two last preconditions. Because *tcb* is a schematic variable, it means that for any *tcb*, if it was in state current before, it still is in state current afterwards. The second postcondition is needed for the *current-threads distinct* invariant described below.

Proving the current-threads-preservation theorem presented above required 1000 LOC. This theorem covers the KernelTransition. The proof for the other transitions was trivial because they do not modify any kernel data.

The are multiple reasons for the size of the 1000-LOC proof: (1) It needs to cover every occasion in which the kernel modifies the state of a thread, which happens frequently. (2) In functions that modify thread states, the state modifications depend on the parameters of these functions. This leads to additional preconditions in these functions' Hoare triples, which have to be shown to hold in the Hoare-triple proof of the calling function(s). Such preconditions sometimes propagate up several function-call levels while changing their shape on each level, depending on the abstraction of the operation on a particular level. (3) The proof needs to cover the entire deletion operation, which is a complex, recursive operation on all object types. (4) In all these cases, we need to prove that none of the affected threads are in state current. Often, this is ensured by a check immediately before. Nevertheless, the cases where the check lies further ahead (e.g. multiple function calls or levels apart) considerably add proof complexity, especially in the delete case.

While the current-threads-preservation theorem is needed to prove refinement of seL4::CMK, the property it states is not specific to a clustered multikernel. It would have been likely that this, or similar property, had already been needed and proved before for the uniprocessor version of seL4, and therefore could have been (re)used.

We have now proved that no CPU will ever delete/modify a thread currently running on another CPU. Unfortunately, this does not exclude two CPUs accidentally scheduling and running the same thread, which violates the unsplittable invariants as well. Hence, we need the following second additional invariant.

Definition 6.4.2 (Current-Threads Distinct)

```
cmk_cur_threads_distinct :: (cmk_loc_A_t, cmk_sh_A_t) priv_t ⇒ bool
cmk_cur_threads_distinct ≡ {(floc,
                            sh) | ∀c1 c2.
                                    let ct1 = snd (fst (floc c1));
                                        ct2 = snd (fst (floc c2))
                                    in st_tcb_at current ct1 sh ∧
                                       st_tcb_at current ct2 sh ∧ c1 ≠ c2 ⟶
                                       ct1 ≠ ct2}
```

For the concrete level, cmk_cur_threads_distinct′ is defined accordingly.
The *current-threads-distinct* invariant states that no two CPUs are running the same (non-idle) thread. Maintaining this invariant requires that the kernel never schedules a thread which is currently running on another CPU. This is exactly the second postcondition of the current-threads-distinct theorem mentioned earlier ($tcb \neq$ cur_thread s). This postcondition ensures that the tcb in question is not scheduled to run on the current CPU (because it is already running on another CPU). Hence, we leverage this postcondition in order to prove the current-threads-distinct invariant, which covers the KernelTransition. Covering the other transitions and proving that the invariant is established required 200 LOC.

6.4.3 Inter-Processor Interrupts (IPIs)

Currently, we do not use IPIs between CPUs within a node to speed up remote thread modification/deletion. If using IPIs is desired for performance reasons, implementing them would be straight-forward. On the formal side, receiving IPIs is already covered by the support for receiving IRQs. Sending IPIs can be modelled as an *abstract machine operation*, a mechanism seL4 already uses for modelling hardware operations. A model like this allows refinement to be proved, i.e. that both levels execute the same abstract machine operation under the same circumstances. However, it does not allow proving properties such as sent IPIs actually being received on the correct CPU. In any case, there would not be much benefit in proving this. As mentioned before, IPIs are not necessary for correctness, only for performance. And showing improved performance is commonly done with benchmarks, not proofs.

6.4.4 Missing Correspondence Proofs

Note that we fixed the running-thread problem in the C implementation and the abstract specification. Applying the fix to the concrete specification and fixing the correspondence proofs is left for future work (Section 7.1.1).

6.4.5 Summary

We have solved the running-thread problem and proved that the solution prevents the kernel from modifying/deleting threads that are currently running at user level on other CPUs. This allows us to prove that the lifted Step function preserves the lifted unsplittable invariants, which is required in order to be able to apply the invariant lifting theorem.

Note that it would be possible to prove additional properties such as: "A thread that is requested to become inactive will in fact become inactive on the next kernel entry", or "the final outcome of a thread modification/deletion is the same regardless of whether the thread was already inactive or had to be requested to become inactive first". However, we consider proving such properties out of scope of this thesis since they are not required for refinement lifting.

6.5 Clustered-Multikernel Refinement Theorem

With the running-thread problem solved, we are finally able to apply the invariant lifting theorem with separation and additional invariants (Theorem 6.2.15). In this theorem, we instantiate I_u with orig_invs_unsplit and I_a' with cmk_cur_threads_distinct. We do this for both abstract and concrete levels.

Note that the theorem's first assumption (invariants hold in the original ADT) was one of the key lemmas we proved in Section 5.10 (Lemma 5.10.3).

The application of the invariant lifting theorem results in the following lemma.

Lemma 6.5.1 *For each node, the invariants (original and additional) hold in the abstract and concrete parallel ADT.*

$$\frac{node\_id < \text{num\_nodes}}{\text{cmk\_ADT\_A } node\_id \models \text{lift\_invs full\_invs cmk\_wrap\_A} \cap \text{cmk\_cur\_threads\_distinct}}$$

$$\frac{node\_id < \text{num\_nodes}}{\text{cmk\_ADT\_C } node\_id \models \text{lift\_invs full\_invs}' \text{ cmk\_wrap\_C} \cap \text{cmk\_cur\_threads\_distinct}'}$$

Using this lemma and Lemma 6.3.8 in the assumptions of Theorem 2.2.9 yields the following lemma.

Lemma 6.5.2 *For each node,* cmk_ADT_C *forward-simulates* cmk_ADT_A.

$$\frac{node\_id < \text{num\_nodes}}{\text{cmk\_ADT\_C } node\_id \sqsubseteq_F \text{ cmk\_ADT\_A } node\_id}$$

After applying Theorem 2.2.6, we get the final refinement theorem for seL4::CMK.

Theorem 6.5.3 (Clustered-Multikernel Refinement) *For each node, the concrete parallel ADT refines the abstract parallel ADT.*

$$\frac{node\_id < \text{num\_nodes}}{\text{cmk\_ADT\_C } node\_id \sqsubseteq \text{ cmk\_ADT\_A } node\_id}$$

80 LOC were needed to prove the theorems three theorems above.

6.6 Assumptions and Limitations

6.6.1 Interleaving Semantics

The lifting operation interleaves the steps of the original automaton non-deterministically. We have used this model to prove refinement of seL4::CMK. Thereby, we have made the assumption that interleaving semantics are applicable. In this section, we informally show why the assumption holds for seL4::CMK.

Recall from Section 2.6.3 that seL4's original ADTs define four transitions: KernelTransition, UserTransition, UserEventTransition and IdleEventTransition. For all combinations of these transitions, we have to show that they either are atomic or do not interfere:

- The UserEventTransition and IdleEventTransition neither interfere with themselves nor with any other transition because they do not access any state. They model the CPU generating a kernel-entry event such as an IRQ, exception or system call.
- The UserTransition is atomic with regards to itself because it allows user level to only modify one byte per step (Definition 2.6.5).
- The KernelTransition is atomic with regards to itself because in the implementation, there is a big lock around kernel entry/exit, which exactly covers the KernelTransition. We further comment on the big lock in Section 6.6.2.
- The only remaining combination is KernelTransition vs. UserTransition. Generally, they do not interfere because they do not access the same state. The only exception is the *IPC buffer*, which is discussed in Section 6.6.3.

6.6.2 Big Lock

Kernel entry/exit code needs to be implemented in assembly because there are no constructs in C that would allow crossing the kernel/user-level boundary. Furthermore, system-call arguments are passed in registers, registers need to be saved and the stack has to be switched. Normally, the register context is saved immediately after having entered the kernel. On x86, there is not even a choice because the CPU automatically saves certain registers when entering kernel mode.

The lock has to be correctly placed within this assembly code. It needs to be acquired before the first access of a shared variable and released after the last such access. Assembly code that only accesses CPU-local state can be left outside the lock. Such code typically initialises CPU registers, configures the CPU or accesses variables on a CPU-local stack (e.g. for saving/restoring CPU registers).

Unlike some other kernels, seL4 configures the CPU not to save the context to a kernel stack, but instead directly to the currently-running thread's TCB, which is in the shared state. Only after the context has been saved, we switch to a proper kernel stack, initialise shared variables and then call the main kernel-entry C function. When the function returns, we do the same in reverse order. Consequently, saving/restoring the register context has to be done outside the lock, which has a potential race. However, this race is eliminated by the solution for the running-thread problem. From Theorem 6.4.1 we know that no thread in state current has its TCB touched by a CPU other than the one it is currently running on. The thread keeps its current state until into the C code, i.e. after the lock is taken. The same applies for kernel exit accordingly.

6.6.3 IPC Buffer

Generally, the seL4 kernel only accesses kernel data, and user-level applications only access user-level frames. This means there is no interference between a KernelTransition and a UserTransition. The only exception is the *IPC buffer*.

For performance reasons, seL4 uses CPU registers to exchange information between user level and kernel. This applies to (1) system-call parameters and return values, (2) fault-handling messages, and (3) payload of IPC messages. However, depending on the CPU architecture, only a small number of CPU registers are available to be used as so called *message registers*. Some system calls have more parameters, some fault types deliver more information and sometimes IPC messages need to be larger than the number of available message registers. This problem is solved by allowing a thread to register a user-level frame as designated IPC buffer. The IPC buffer has a fixed structure—essentially an array of additional word-sized message registers—which is read and written by both kernel and user level.

Nevertheless, a race condition can only occur if the same IPC buffer is used for multiple threads, which clearly is a misconfiguration. Such misconfigurations are possible on the uniprocessor version as well and most likely end in the misconfigured threads crashing due to corrupted system-call parameters or return values. However, it is ensured that only the misconfigured threads are affected. It is not possible to crash the kernel or other isolated user-level components with this misbehaviour. All user-level input, in registers and in the IPC buffer, is sanitised by the kernel. Sanitisation is proven correct in seL4, i.e. the uniprocessor refinement proof holds no matter what the user-level input is.

Problem

At first sight, we are done: While we have some interference between a KernelTransition and a UserTransition in case of misconfigured IPC buffers, the kernel is proven to work correctly, no matter what the outcome of that interference is.

Unfortunately, there is a small subtlety: In the uniprocessor model, the contents of IPC buffers stay the same during a KernelTransition unless the kernel itself modifies them. In seL4::CMK, however, a misconfigured IPC buffer might be modified at user level on one CPU while being read by the kernel on another CPU.

In order to finish the argument that the interleaving model is applicable, it therefore remains to be shown what we call the *IPC-buffer-robustness property*: "For correctness, the kernel does not rely on IPC-buffer contents to never change during a KernelTransition." We aim to show—by code inspection—that this property holds for seL4.

Unfortunately, the property as defined above is hard to capture in a more formal way. It is not immediately clear what the inspected code needs to look like in order to guarantee "correctness of the kernel" in case IPC buffers are modified during a KernelTransition. Hence, we inspect the code for a stronger property that implies the IPC-buffer-robustness property. An example of such a property is the *sequential-semantics property*: "After having read or written a specific memory address in an IPC buffer, this memory address is never read again in the same KernelTransition". This prevents the kernel from reading inconsistent values and from storing values and reading them back later. Combined with the fact that sanitisation is proven correct for the uniprocessor case, this rules out any possible kernel misbehaviour in case IPC buffers are modified concurrently.

Note that it would be possible to slightly adapt the implementation to create a local copy of all message registers on kernel entry and only access these copies during a KernelTransition. This would make code inspection trivial and thus provide a high degree of assurance that the IPC-buffer robustness property holds. However, copying all message registers on kernel entry adds a high performance penalty. We therefore aim for a more complex inspection of the original seL4 code.

A thorough code inspection inevitably has to go into a certain amount of implementation detail. However, we cannot assume the reader is familiar with seL4 internals. Therefore, we first give a high-level view of how system calls are implemented in seL4. Then, we dive into more detail, down to specific C functions. For each function that we name explicitly, we mention its purpose and explain how it interacts with the IPC buffer. Note that it is not necessary to know the entire semantics of these functions in order to follow the argument of the code inspection.

Code Inspection

A system call in seL4 is divided up into three stages:

1. The kernel decodes the system call. This involves reading arguments from the caller's IPC buffer and checking them for sanity. It is also checked if the caller provided the necessary capabilities to perform the system call. In case a check does not succeed, we jump to stage 3 to return an error message to user level.

2. The kernel performs the system call. If it is an IPC transfer, the message is copied from the sender thread's to the receiver thread's IPC buffer. If it is a fault, the kernel sends a fault IPC message to the registered fault-handler thread,[5] which involves

[5]User-level faults in seL4 are handled by a fault-handler thread that is assigned to the faulting thread.

writing to its IPC buffer. Other system calls do not access IPC buffers.

3. The kernel returns the result of the system call to user level. In case of an error, the error message is written to the caller's IPC buffer before returning to user level.

It can be observed that system calls follow a read-modify-write pattern with regards to IPC buffers. In stage 1, the function `getSyscallArg` is used to read system-call arguments that are located in the message registers of the caller's IPC buffer. Our code inspection confirmed that each required argument is read exactly once. The function `lookupExtraCaps` is used to read arguments that contain pointers to capabilities. Message registers and capability-pointer arguments are located in two separate arrays in the IPC buffer. Consequently, no IPC buffer address is read twice during stage 1. So far, the sequential-semantics property holds.

For stage 2, we need to inspect four different system calls that access IPC buffers:

1. In case of an IPC transfer, the kernel calls the function `copyMRs`, which copies the message from the sender thread's to the receiver thread's IPC buffer. Each message register is copied exactly once. None of these message registers have been read by `getSyscallArg` before because this system call has no arguments. Even if both threads, sender and receiver, have registered the same IPC buffer (misconfiguration), the sequential-semantics property is not violated because every message register is first read and then written.

   If a capability needs to be transferred as well, the kernel calls `getReceiveSlot`, which reads the destination capability pointer from the receiver's IPC buffer. It is located in a separate field in the IPC buffer and therefore cannot have been read before by `getSyscallArg`, `lookupExtraCaps` or `copyMRs`.

   In case a so-called *badge-unwrap*[6] operation is required, the kernel additionally calls `setExtraBadge`, which writes the unwrapped badge to the receiver thread's IPC buffer. More specifically, it writes to the same array from which `lookupExtraCaps` has read the capability-pointer arguments in stage 1. However, since we write after we read, we do not violate the sequential-semantics property.

2. In case of an asynchronous IPC transfer, the kernel writes one message register to the receiver thread's IPC buffer. The sequential-semantics property is still preserved since we only write.

3. In case of a fault, the kernel calls `setMRs_fault`, which writes the fault message to the message registers in the fault-handler thread's IPC buffer. The sequential-semantics property is still preserved since we only write.

4. In case a fault-handler thread replies to an unknown-system-call fault,[7] the kernel calls `handleFaultReply`, which reads message registers from the fault-handler thread's IPC buffer. Each message register is read exactly once. None of these message registers have been read by `getSyscallArg` before because this system call has no arguments.

Consequently, after stage 1 and stage 2, the sequential-semantics property still hoods. In stage 3, the kernel returns the result of the system call to user level. In case of an error,

[6] An unwrapped badge is a number that identifies the sender of an IPC transfer.

[7] Unknown-system-call faults can be used in a virtualisation setup to virtualise system calls of a guest OS.

the error message is written to the caller's IPC buffer before returning. The sequential-semantics property is preserved because we only write.

We have therefore shown—by code inspection—that for each system call, the sequential-semantics property holds.

Note, however, that seL4 also provides a meta system call named *ReplyWait*, which does nothing else than executing the *Reply* and then the *Wait* system call within a single KernelTransition. It primarily exists for performance reasons because in most use cases, Wait is executed immediately after Reply. The problem now arises that, while both system calls individually satisfy the sequential-semantics property, executing them in sequence within a single KernelTransition does not. The Reply system call might write to the same IPC buffer which the Wait system call later reads from. However, we argue that this cannot cause any problems because the ReplyWait meta system call invokes the Reply and Wait system calls at the outermost level of the KernelTransition and each of these system calls is self-contained. Therefore, concurrent changes to an IPC buffer after Reply has finished and before Wait has started do not cause any problems.

Alternatively, the ReplyWait meta system call could also be disabled in the kernel and emulated at user level with a slight performance penalty.

In summary, we have shown that the IPC-buffer-robustness property holds, which concludes the informal proof of the assumption that interleaving semantics are applicable when applying the refinement lifting operation to the global automaton of seL4.

6.7 Conclusion

In this chapter, we presented the part of the refinement lifting framework that is concerned with the runtime phase of the kernel. For this phase, the kernel isolation theorem (Section 5.7) allows us to reason about each node in isolation. Consequently, this chapter took a node-local view.

In seL4's refinement proof, the ADTs define a *global automaton* (Section 2.6.3), which models the actions of the kernel on a single CPU. Within a node of a clustered multikernel, we have multiple CPUs running in parallel. We therefore modelled a node as a parallel composition of the original automaton.

Section 6.2 contributed a formal *lifting operation* with accompanying *refinement lifting theorem* and *invariant lifting theorem*. The job of the lifting operation is to turn the original model into a valid model of a node running on multiple CPUs. It does this by lifting the original automaton into a parallel composition of itself with a parameterisable splitting of the original state into shared and local parts. Steps of multiple CPUs are interleaved non-deterministically. The lifting operation also defines what refinement means in this context. We assume that interleaving semantics are applicable.

To show the practicability of the lifting operation and theorems, we applied them to seL4 (Section 6.3). This involved instantiating the lifting operation's type variables, defining the parameterisation functions and separating seL4's unsplittable invariants.

We have proved (1) the required properties about the parameterisation functions, (2) that the splittable invariants are splittable, and (3) that the unsplittable invariants are established. However, proving that the unsplittable invariants are preserved required solving the *running-thread problem* (Section 6.4). With this problem solved, we were able to prove and present our goal, the *clustered-multikernel refinement theorem* (Section 6.5).

Finally, Section 6.6 identified that the lifting operation relies on the assumption that interleaving semantics apply and showed—by informal argument and code inspection—that this assumption is correct for seL4.

Chapter 7

Conclusion

OS kernel correctness is crucial for secure, safe and reliable computer systems. Strong kernel correctness guarantees can be obtained by formal verification down to the implementation level. In order to keep verification complexity at a manageable level, prior kernel verification research only targeted uniprocessor kernels. The lack of verified multiprocessor kernels is a problem because manufacturers are increasing the computing power of their systems by adding more CPUs and cores.

In this thesis, we demonstrated that it is possible to add multiprocessor support to a verified uniprocessor kernel and leverage the existing proofs to obtain a verified multiprocessor version of that kernel.

To this end, we introduced the *clustered multikernel*, a point in the design space of multiprocessor kernels. The main feature of this design is that it reduces concurrent data access to a minimum while offering a configurable trade-off between scalability and flexibility. This is possible by confining the required concurrent data accesses such that reasoning about them can be decoupled from reasoning about the kernel's functionality. Furthermore, the design eases conversion of a uniprocessor kernel into a multiprocessor kernel. For this purpose, we presented a *conversion scheme* to convert a uniprocessor kernel into a clustered multikernel.

Based on this design, we contributed a *refinement lifting framework*, which lifts the converted kernel's functional-correctness proof such that it applies to the clustered-multikernel version (under certain assumptions). The support for handling the introduced concurrency was added to the existing verification framework in a non-intrusive way. The refinement lifting framework accounts for weak memory ordering exhibited by total-store-order (TSO) multiprocessor architectures.

We demonstrated the practicability of the conversion scheme and the refinement lifting framework by applying them to seL4, a general-purpose uniprocessor microkernel, which was formally verified in a large-scale research project [KEH+09]. This makes seL4::CMK the general-purpose multiprocessor kernel with the highest degree of formal assurance.

The lifting framework is applicable to other kernels as well, under the condition that refinement of these kernels has been proved via forward simulation with the calculus presented in Section 2.2 and that interleaving semantics apply (as explained in Section 6.6).

We showed that our approach requires relatively low effort, compared to the kernel's initial verification: The proofs that were required for the refinement lifting framework and its application to seL4's first refinement step comprise about 9 kLOC in total. For a full implementation-level proof, we have to add a few kLOC for the missing correspondence proofs and second refinement step (Section 7.1.1 and Section 7.1.2). Compared to L4.verified's overall proof size of 200 kLOC [KEH+09], this is relatively small.

We conclude that for verified uniprocessor kernels that required a large initial proof effort, our approach provides an attractive way to obtain a verified multiprocessor version of that kernel with relatively low effort. On the other hand, for very small and simple verified kernels, it might be a better approach to directly reimplement and reprove a multiprocessor version.

7.1 Future Work

7.1.1 Missing Correspondence Proofs in First Refinement Step

The refinement lifting framework concentrates on the abstract level and relies on the correspondence proofs to transfer the necessary theorems down to the concrete level. For seL4::CMK, two correspondence proofs are missing.

The first missing correspondence proof belongs to the solution of the running-thread problem (Section 6.4).

The second one is the correspondence proof of kernel bootstrapping, as mentioned in Section 5.11.3. However, note that recently, an abstract and an intermediate specification have been written for uniprocessor seL4's bootstrapping, based on the abstract specification of seL4::CMK's bootstrapping presented in this thesis. The missing uniprocessor bootstrapping correspondence proofs are currently being worked on by the seL4 verification team. When finished, they need only be slightly adapted to cover part 2 of seL4::CMK's bootstrapping. Part 1 of bootstrapping needs correspondence proofs of its own.

The missing correspondence proofs can be carried out in the same way and with the same verification framework they had been carried out in seL4's past. There is nothing concurrency-specific in them.

7.1.2 Second Refinement Step

In order to extend the refinement lifting down to seL4's C implementation, the refinement lifting framework has to be formally applied to seL4's second refinement step as well. We expect the involved effort to be much smaller than for the first refinement step for the following reasons.

The kernel-memory-sequential-access and isolation theorems do not have to be proved again on the intermediate level. As soon as the missing bootstrapping correspondence proof is done, these theorems transfer down from the abstract to the intermediate level.

The running-thread problem does not have to be solved again. The additional invariants it required are proved on the abstract level and transfer down to the intermediate level as soon as the correspondence proof of the running-thread problem's solution is fixed.

7.1.3 Enhancements to the TSO Model

Recall from Section 5.2.2 that our TSO model does not store the actual contents of memory. However, there is no conceptual barrier to adding this to the model. Combined with additional abstract instructions working with these contents, this would make it possible to reason about parallel programs using variables to synchronise concurrent actions. However, the drawback of using the TSO model to store program state is that when reading/writing abstract objects, their content has to be mapped to/from byte values.

This defeats one of the main purposes of abstract specifications: being able to abstract away from low-level representation of data.

We believe that a good compromise is to divide up the program state into abstract objects and synchronisation variables. This division should be possible for most parallel programs unless program logic is too intermingled with synchronisation (e.g. lock-free data structures). The synchronisation variables on the level of the TSO model can then be used to reason about concurrency, e.g. to prove deadlock freedom or that when accessing abstract objects in critical sections, sequential semantics are observed. It would also be possible to directly model architecture-provided synchronisation primitives such as CAS (compare-and-swap) or TAS (test-and-set).

7.1.4 Resource Transfer Between Nodes

The design of the clustered multikernel presented in Chapter 4 requires complete isolation of kernel memory between nodes. In case of seL4, this includes untyped memory, from which kernel objects are created. Kernel resource balancing between nodes is therefore impossible because—due to the restriction—untyped memory cannot be moved between nodes. If one node needs to create more kernel objects and runs out of untyped memory, it is not possible to obtain it from another node who might have enough untyped memory. The problem is fundamental in the sense that a clustered multikernel based on a kernel other than seL4 would suffer from the same problem as well.

The reason for completely isolating kernel memory is because it ensures that no kernel data is ever accessed concurrently by two nodes. However, in order to achieve this, complete isolation is a stronger restriction than immediately necessary. It would be possible to relax it and allow kernel memory to be transferred between nodes, as long as it is ensured that it will only be accessed by the current owner.

For a capability-based kernel such as seL4, this means that an object can only be transferred to another node if it currently has no dependencies. Unfortunately, in a running system, most objects do have dependencies. Examples are page tables, mapped frames and TCBs. It is therefore not possible to transfer address spaces or threads. However, there are two types of memory that can be freed from dependencies if desired: unmapped frames and untyped memory.[1]

Unmapped frames are user-level frames that are currently not mapped into a virtual address space. They are free of dependencies if there are no references (anymore) to the untyped memory the frames were created from. It is perfectly possible to design a system where this is the case. The only drawback is that the memory these frames occupy cannot be used anymore for anything else than frames. Transferring unmapped frames would enable data transfer between nodes without having to copy to/from a shared memory region.

Untyped memory is free of dependencies as long as no kernel objects exist that have been created from it. Transferring untyped memory therefore enables kernel resource balancing, i.e. balancing of free kernel memory between nodes.

In order to maintain continuous isolation between nodes, the sending node has to ensure that any unmapped frame or untyped memory it transfers is free of dependencies. This can be done with a simple check before the transfer. Formally, it has to be proved that after the transfer, no reference to the transferred memory exists anymore in the sending node. Without a reference, the memory cannot be accessed anymore.

[1] Please refer to Section 2.5.1 for the definition of *untyped memory*.

For the transfer itself, the sending and receiving node need a rendez-vous point, most likely implemented in a pair of send/receive system calls (or methods in a capability-based kernel). Furthermore, a small region of shared memory is needed which contains the information about the transferred memory (type, address, size). With appropriate encoding, this region of memory need not be larger than a single machine word. The problem is that this region of memory (or machine word) needs to be accessed concurrently by two nodes during the kernel's runtime phase, something the refinement lifting framework does not support yet.

The reason this is not supported is that for the runtime phase of the kernel, the refinement lifting framework takes a completely node-local view. It is therefore not possible to formulate (and prove) properties that talk about memory that is accessed by two nodes, e.g. a sending and a receiving node.

To circumvent this problem, we could model the transfer of the required machine word as an abstract machine operation, instead of a concurrent memory access. The transfer of this word would be implemented "behind the scenes", possibly with a hidden global variable. Only the interface of that abstract operation (send/receive) would be visible externally. This would allow formally specifying both parts of the transfer (sending and receiving) individually. It would also allow proving that transferred memory will not be accessed anymore by the sending node. Nevertheless, we still could not prove properties that talk about both the sending and receiving part. For example, we could not formulate a theorem saying that for a particular transfer, the receiver receives exactly what the sender has sent.

7.1.5 Performance/Scalability Evaluation of the Clustered Multikernel

In Chapter 4, we have introduced the clustered-multikernel design and discussed its implications on systems design, performance and scalability from a theoretical point of view. However, it would be desirable to know in detail how the design performs in practice. Therefore, we propose a performance and scalability evaluation of the clustered-multikernel design based on benchmarks of seL4::CMK.

Performance Questions

System designers find themselves in the following situation: Building a computer system on top of a multiprocessor kernel that employs traditional synchronisation mechanisms (e.g. fine-grained locks, lock-free) gives them (1) good scalability and (2) flexible kernel-resource usage across CPUs *at the same time*. However, when requiring a verified multiprocessor kernel, they are restricted to a clustered multikernel where they need to trade scalability for flexibility.

Therefore, the evaluation should answer the following questions: How do performance/scalability of a given workload compare when running it on a clustered multikernel vs. running it on a multiprocessor kernel employing traditional synchronisation mechanisms? Which cluster configurations yield the best results? How do these configurations relate to the underlying hardware?

Note that the evaluated workload needs to be configurable in terms of clustering. More precisely, it must be possible to distribute it across nodes (to benchmark configurations with multiple nodes), and it must be possible to run it multi-threaded (to benchmark configurations with multiple CPUs per node).

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, October 2006.

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[ABK12] June Andronick, Andrew Boyton, and Gerwin Klein. Formal system verification — Extension. Technical Report 1833-9646-6408, NICTA, Sydney, Australia, October 2012.

[AHL+08] Eyad Alkassar, Mark Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2008.

[AHPP10] Eyad Alkassar, Mark Hillebrand, Wolfgang Paul, and Elena Petrova. Automated verification of a small hypervisor. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2010*, volume 6217 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2010.

[AMDa] AMD Developer Central. IPI-reordering question. `http://devgurus.amd.com/thread/134043`.

[AMDb] AMD, Inc. AMD64 architecture programmer's manual. `http://developer.amd.com/Resources/documentation/guides/Pages/default.aspx#manuals`.

[Bac90] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer, 1990.

[BBBB09] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability and Security*, Hamburg, Germany, 2009.

[BBBB10] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Ingredients of operating system correctness. In *Embedded World Conference*, Nuremberg, Germany, March 2010.

[BBBT11] Christoph Baumann, Thorsten Bormer, Holger Blasum, and Sergey Tverdyshev. Proving memory separation in a microkernel by code level verification. In *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 25–32, Newport Beach, CA, USA, 2011. IEEE Computer Society.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009. ACM.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, Amsterdam, The Netherlands, 2006. Springer.

[BDR97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France, October 1997.

[Bev89] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together—formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):411–430, 2006.

[BLW08] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie — an interactive prover for the Boogie program-verifier. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2008.

[BMSW10] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie — an interactive prover-backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, February 2010.

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, May 2009.

[BWCC+08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many

cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, USA, December 2008.

[BWCM+10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–8, Vancouver, BC, Canada, 2010.

[BWKMZ12] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the 2012 Ottawa Linux Symposium*, Ottawa, Canada, July 2012.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer.

[CHP+08] Juan Chen, Chris Hawblitzel, Frances Perry, Mike Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 183–192, Tucson, AZ, USA, 2008. ACM.

[CKS08] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182, Montreal, Canada, August 2008. Springer.

[CRD+95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, USA, December 1995.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 51–65, Pittsburgh, PA, USA, 2008. IEEE Computer Society.

[CS09] Ernie Cohen and Norbert Schirmer. A better reduction theorem for store buffers. *Computing Research Repository*, abs/0909.4637, 2009. `http://arxiv.org/abs/0909.4637v1`.

[CS10] Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. In Matt Kaufmann, Lawrence Paulson, and Michael Norrish, editors, *International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science, Edinburgh, UK, July 2010. Springer.

[CYC+01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[DDW09] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, 42(2–4):349–388, 2009.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[dRT08] Tom In der Rieden and Alexandra Tsyban. CVM — A verified framework for microkernel programmers. In *Proceedings of the 3rd Systems Software Verification*, Electronic Notes in Computer Science, pages 137–153, Sydney, Australia, February 2008. Elsevier.

[DSS09] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation correctness of a real-time operating system. In *IEEE International Conference on Software Engineering and Formal Methods*, pages 23–32, Hanoi, Vietnam, 2009. IEEE Computer Society.

[EDE08] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 35–40, Glasgow, UK, April 2008. ACM SIGOPS.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, USA, December 1995.

[FN79] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conference Proceedings, 1979 National Computer Conference*, pages 329–334, New York, NY, USA, June 1979.

[Fre06] Torsten Frenzel. Design and implementation of the L4.sec microkernel for shared-memory multiprocessors. Diploma thesis, TU Dresden, Germany, August 2006. `http://os.inf.tu-dresden.de/papers_ps/frenzel-diplom.pdf`.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The Sawmill multiserver approach. In *Proceedings of the 9th SIGOPS European Workshop*, pages 109–114, Kolding, Denmark, 2000. ACM Press.

[GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, New Orleans, LA, USA, February 1999.

[GNU] GNU GRUB. `http://www.gnu.org/software/grub`.

[Har85] Norman Hardy. KeyKOS architecture. *ACM Operating Systems Review*, 19(4):8–25, October 1985.

[Hat97] Les Hatton. Re-examining the fault density - component size connection. *IEEE Software*, 14(2):89–97, 1997.

[Her10] Jorrit N. Herder. *Building a Dependable Operating System: Fault Tolerance in MINIX 3*. PhD thesis, Department of Computer Science, Vrije Universiteit Amsterdam, September 2010.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, 2001.

[HLSD+13] Gernot Heiser, Etienne Le Sueur, Adrian Danis, Aleksander Budzynowski, Tudor-Ioan Salomie, and Gustavo Alonso. RapiLog: Reducing system complexity through verification. In *Eurosys Conference (EuroSys)*, page 14, Prague, Czech Republic, April 2013.

[Hoh02a] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, Germany, July 2002.

[Hoh02b] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, Dresden University of Technology, July 2002.

[HOL] HOL4 Theorem Prover. `http://hol.sourceforge.net`.

[HT03] Michael Hohmuth and Hendrik Tews. The semantics of C++ data types: Towards verifying low-level system components. In *Emerging Trends Track, TPHOLs*, Rome, Italy, September 2003.

[HT05] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd Workshop on Programming Languages and Operating Systems (PLOS)*, Glasgow, UK, July 2005.

[IBM05] IBM. PowerPC architecture book, version 2.02, November 2005. `http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html`.

[Inta] Intel Corporation. Intel 64 and IA-32 architectures software developer manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[Intb] Intel Developer Zone. IPI-reordering question. `http://software.intel.com/en-us/forums/topic/289491`.

[K42] K42/Tornado Operating System. `http://www.eecg.toronto.edu/~tornado`.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[KMG+11] Gerwin Klein, Toby Murray, Peter Gammie, Thomas Sewell, and Simon Winwood. Provable security: How feasible is it? In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, pages 28–32, Napa, CA, USA, May 2011. USENIX.

[KSW10] Gerwin Klein, Thomas Sewell, and Simon Winwood. *Refinement in the formal verification of seL4*, pages 323–339. Springer, March 2010.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification—from verified programs to verified systems. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd Systems Software Verification*, volume 217 of *Electronic Notes in Computer Science*, pages 23–40, Sydney, Australia, February 2008. Elsevier.

[Lyo11] Anna Lyons. Efficient concurrency control for high-performance microkernels. BSc thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, July 2011. Available from publications page at `http://ssrg.nicta.com.au/`.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronisation on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[MMB+12] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 126–142. Springer, December 2012.

[MMB+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2013.

[mos] mosbench scalability benchmark. `http://pdos.csail.mit.edu/mosbench`.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, NV, USA, October 1998.

[NBF+80] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Larry Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, May 1980.

[NF03] Peter G. Neumann and Richard J. Feiertag. PSOS revisited. In *19th Annual Computer Security Applications Conference (ACSAC)*, pages 208–216, Las Vegas, NV, USA, December 2003.

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NW77] R.M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pages 1–10. ACM, November 1977.

[OKL12] Open Kernel Labs software surpasses milestone of 1.5 billion mobile device shipments. `http://www.ok-labs.com/releases/release/ok-labs-software-surpasses-milestone-of-1.5-billion-mobile-device-shipments`, January 2012. Media Release.

[Ope] Open Kernel Labs. OKL4 Microvisor. `http://www.ok-labs.com/products/okl4-microvisor`.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Munich, Germany, 2009. Springer.

[PD97] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.

[Pot99] Daniel Potts. L4 on uni- and multiprocessor Alpha. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1999. Available from publications page at `http://ssrg.nicta.com.au/`.

[PSS12] Wolfgang Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor — assembler code verification. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2012.

[PWH02] Daniel Potts, Simon Winwood, and Gernot Heiser. Design and implementation of the L4 microkernel for Alpha multiprocessors. Technical Report UNSW-CSE-TR-0201, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, February 2002.

[QNX] QNX Software Systems. QNX Neutrino RTOS. `http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html`.

[Rid10] Tom Ridge. A rely-guarantee proof system for x86-TSO. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2010*, volume 6217 of *Lecture Notes in Computer Science*, pages 55–70, Edinburgh, UK, 2010. Springer.

[Rit93] Duncan Stuart Ritchie. The Raven kernel: a microkernel for shared memory multiprocessors. Technical Report TR-93-36, University of British Columbia, Vancouver, BC, Canada, April 1993.

[RN93] D. Stuart Ritchie and Gerald W. Neufeld. User level IPC and device management in the Raven kernel. In *Proceedings of the 2nd USENIX Workshop on Microkernels and other Kernel Architectures*, pages 111–125, San Diego, CA, USA, September 1993.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[Sch11] Mareike Schmidt. *Formal Verification of a Small Real-Time Operating System*. PhD thesis, Saarland University, Saarbrücken, April 2011.

[SDN+04] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification*, Sydney, Australia, October 2004.

[SK10] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*, Paris, France, April 2010.

[SMK13] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, USA, June 2013. ACM.

[SPA92] SPARC International Inc. The SPARC architecture manual, version 8, 1992. `http://www.sparc.org/standards/V8.pdf`.

[SPB+08] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the 1st Workshop on Managed Many-Core Systems*, Boston, MA, USA, June 2008.

[SPE00] SPEC CPU2000 benchmark. `http://www.spec.org/cpu2000`, August 2000.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Charleston, SC, USA, December 1999.

[SSGA11] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the 6th EuroSys Conference*, pages 17–30, Salzburg, Austria, April 2011. ACM.

[SSN+09] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Savannah, GA, USA, 2009.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[ST08] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd Systems Software Verification*, volume 217 of *Electronic Notes in Computer Science*, pages 169–185, Sydney, Australia, February 2008. Elsevier.

[SWG+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *2nd International Conference on Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer.

[TWV08] Hendrik Tews, Tjark Weber, and Marcus Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd Systems Software Verification*, volume 217 of *Electronic Notes in Computer Science*, pages 79–96, Sydney, Australia, February 2008. Elsevier.

[Uhl05] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Karlsruhe, Germany, June 2005.

[UKGS93] Ron Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9:105–134, 1993.

[Vera] Verisoft Project. `http://www.verisoft.de`.

[Verb] Verisoft XT Project. http://www.verisoftxt.de.

[VMw09] VMware. Performance evaluation of Intel EPT hardware assist, 2009. http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[vT10] Michael von Tessin. Towards high-assurance multiprocessor virtualisation. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *Proceedings of the 6th International Verification Workshop*, volume 3 of *EasyChair Proceedings in Computing*, pages 110–125, Edinburgh, UK, July 2010. EasyChair.

[vT12] Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *Proceedings of the 2nd Workshop on Systems for Future Multi-core Architectures*, Bern, Switzerland, April 2012.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[YH10] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, Toronto, Ontario, Canada, June 2010. ACM.