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

Michael von Tessin
NICTA* and University of New South Wales
Sydney, Australia
michael.vontessin@nicta.com.au

ABSTRACT

Operating-system kernels are critical software components in computer systems. Building secure, safe and reliable computer systems is facilitated by having strong kernel correctness guarantees. Such guarantees can be obtained by formally verifying a kernel down to the implementation level. Kernel verification has attracted much research interest. For example, the L4.verified project has proved that the implementation of the seL4 microkernel adheres to its formal specification. Nonetheless, due to verification complexity, past research focussed on uniprocessor kernels only. However, with multiprocessor/multicore systems gaining popularity, also in embedded systems, the need for verified multiprocessor kernels arises.

To this end, we introduce the *clustered multikernel*, a point in the design space of multiprocessor kernels. Based on this design, we present a *lifting framework*, which adds multiprocessor support to a verified uniprocessor kernel and reuses its proofs to obtain a verified multiprocessor kernel with relatively low effort. The lifting framework supports total-store-order (TSO) multiprocessor architectures, which exhibit weak memory ordering. We report on our experience with applying the lifting framework to seL4.

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

1. INTRODUCTION

Operating-system (OS) kernels are critical software components in computer systems. According to Hatton [10], "good" software contains 6 bugs per thousand lines of code (LOC) on average and with "our best techniques" we can achieve 0.5–1 bugs per kLOC. This is not enough for kernels in high-assurance computer systems used, for example, in defence, aviation and the like.

Strong correctness guarantees can be obtained by formally verifying a kernel down to the implementation level. The history of kernel verification starts in the 70s and 80s [2,20], but none of these early attempts "produced a realistic OS kernel with full implementation proofs" [11]. In the first decade after 2000 the topic attracted new interest. Verifying a kernel was part of several verification projects [7, 8, 12, 17]. The largest verified kernel is seL4 [12], an 8700-LOC general-purpose microkernel, which was verified down to C 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, which is typically introduced by (1) switching between multiple threads of execution (kernel threads, interrupt handlers) or (2) running multiple CPUs in parallel. 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[1]).

In summary, the current state of research restricts computer systems that require a verified OS kernel to running only on one CPU. This is a problem because manufacturers increase 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.

Verifying an kernel down to implementation level requires a large effort, which often includes the development of the verification framework. For example, verifying the seL4 microkernel required 20 person years (py) of which 11 py were kernel-specific efforts [12]. The rest went into developing the verification framework, proof libraries etc. Therefore, it is desirable to leverage as much as possible from an existing verification framework and kernel proof.

To this end, we introduce the *clustered multikernel*, a point in the design space of multiprocessor kernels. Based on this design, we present a *lifting framework* which adds multiprocessor support to a verified uniprocessor kernel and reuses its proofs to obtain a verified multiprocessor kernel with relatively low effort. The ability to handle the concurrency introduced with multiple CPUs is added to the existing verification framework in a modular and non-intrusive way. The lifting framework supports total-store-order (TSO) multiprocessor architectures, which exhibit weak memory ordering. For this purpose, the formal TSO model which it builds on explicitly models CPUs starting other CPUs (also nested) in presence of memory reordering.

To demonstrate the practicability of the lifting framework, we report on our experience with applying it to seL4.

All formal specifications and proofs in this work are machine-checked in the interactive theorem prover Isabelle/HOL [14].

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

[1]The term *CPU* is used as synonym for *core* hereafter. The same is true for the terms *multiprocessor* and *multicore*.

The paper continues as follows: Section 2 summarizes background information and introduces terminology related to formal verification. Our main results—the multikernel design, the lifting framework and its application—are described in Section 3 and Section 4, respectively. We relate this paper to other work in Section 5 and conclude in Section 6.

2. BACKGROUND AND TERMINOLOGY

Formal verification of concurrent systems is hard and becomes intractable very quickly. In case of a *model-checking* approach where all possible states are enumerated and checked, the complexity grows exponentially with regards to program size and number of execution units (CPUs on kernel level). This scalability problem can be overcome with a *theorem-proving* approach where proofs only have to cover all possible conceptual "scenarios" which can arise from concurrent execution. The number of such scenarios is a fraction of the number of states in model checking and can be lowered even further with intelligent proof design. Nevertheless, the nature of theorem proving requires considerable human input to prove each of these scenarios.

Functional correctness is proved by *refinement*, i.e. we prove that an implementation *refines* a specification. A refinement proof is based on a *refinement automaton*. This is a non-deterministic finite state automaton with *transitions* from one *state* to another. It describes how the software component in question reacts to specific *events*, for example, how a kernel reacts to system calls, faults or interrupts depending on the state it is in (e.g. kernel/userlevel/idle). Initialisation of the automaton is done by an *initialisation function*, which, in the case of a kernel, models that kernel's *bootstrapping phase*. "Running" the automaton models the *runtime phase* of the kernel.

Proving refinement consists of mainly two efforts: One is proving *correspondence* between the concrete and abstract counterparts of each transition and of the initialisation function. These proofs normally require certain *invariants* to hold. Hence, the other effort is to prove that all these invariants hold.

A refinement proof *transfers* safety theorems proved on the abstract level down to the concrete level. We leverage that fact by proving the necessary theorems on the abstract level only and let the refinement transfer them to lower levels where proving them would be much harder.

Verification frameworks used to verify uniprocessor kernels are normally designed for sequential reasoning only as the need to reason about concurrency can be avoided by making a kernel non-preemptible or event-based with well-defined preemption points.

There are many different ways how a verification framework can support concurrency. Mostly, they are based on an interleaving model. However, such models are problematic in a refinement context because of the atomicity mismatch of abstract and concrete operations. Furthermore, concurrent abstract specifications are hard to reason about, which conflicts with the goal of an abstract specification: to make reasoning about a program simpler.

As a consequence, the main design goal of the lifting framework is to reduce the amount of concurrency we have to reason about to a minimum.

3. THE CLUSTERED MULTIKERNEL

There are two fundamental ways to avoid reasoning about concurrency: (1) to avoid parallelism (i.e. running things in parallel), or (2) to avoid sharing (of data structures).

We can avoid parallelism in a kernel by having a *big lock* around the whole kernel, which only allows one CPU to perform a kernel transition at any given time. As userlevel code is allowed to run in parallel on multiple CPUs, kernel calls can become a bottleneck.

On the other side of the spectrum, we can avoid sharing with a restricted *multikernel* [1] design in which each CPU boots up its own *node*. A node is an isolated instance of a uniprocessor kernel including userlevel running on top. As a consequence of the isolation, no communication between the node's kernels can exist and no kernel data structures can be shared or moved between nodes. Nevertheless, a multikernel can be bootstrapped with provision for a designated region of shared userlevel memory between nodes.

While the multikernel does not have the scalability problem of the big-lock design, it lacks the latter's ability to make the step to multiple CPUs transparent to userlevel. A multikernel application has to deal with the consequences of multiple nodes, such as explicit userlevel memory sharing and kernel resources being bound to nodes. Furthermore, as inter-process communication (IPC) only works within a node, communication across nodes has to be implemented via shared userlevel memory.

In summary, the *big-lock* design has low scalability but high flexibility while the opposite is true for the *multikernel* design. Of course, we want the best of both, which means we want the trade-offs to be configurable according to the type of userlevel application. This is possible with a *clustered multikernel*, which is a configurable combination of the two designs (Figure 1). It starts out as a multikernel but instead of running one node per CPU, multiple CPUs can be *clustered* into (assigned to) a node. Within each node, we apply the big-lock design to synchronise its CPUs.

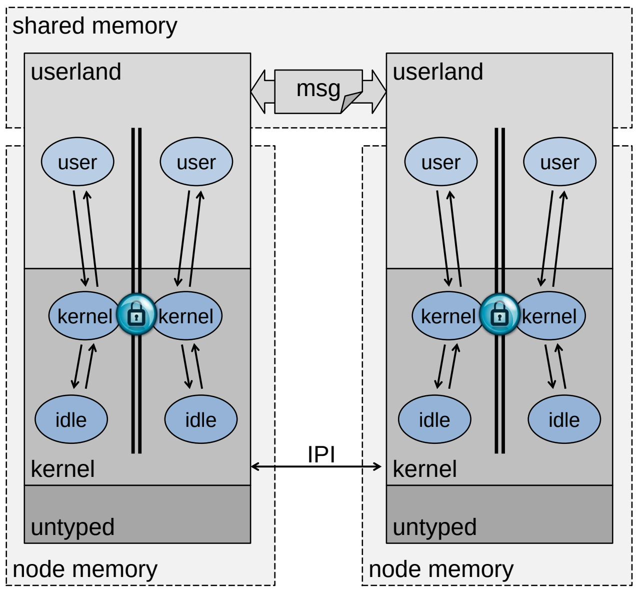

Figure 1: The Clustered Multikernel

The flexibility in clustering allows a kernel to be configured with the underlying hardware in mind. For example, multiple cores within a CPU package can be clustered into a node because 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".

The isolation between nodes also allows clustering to be used to draw performance-isolation boundaries for real-time systems. Inside the performance-isolated domains, we can leverage the flexibility of a big-lock kernel.

The clustered multikernel resembles the clustered kernels of Hurricane [18] and Hive [5] which are discussed in Section 5.

4. LIFTING FRAMEWORK

The lifting framework turns a verified uniprocessor kernel into a clustered multikernel and formally lifts the uniprocessor refinement proof into the clustered-multikernel context. Lifting a proof means reusing the proved theorem in a more generic context. For example, a theorem about a kernel-internal function can be directly reused if the lifting framework ensures that no concurrency is introduced for that particular function.

The main challenges for the lifting framework are: (1) how to correctly handle the concurrency that cannot be avoided in the clustered multikernel; (2) how to non-intrusively extend a given uniprocessor verification framework; and (3) how to reuse as much as possible from the uniprocessor proof.

To this end, the lifting framework prescribes: (1) a recipe to turn a uniprocessor kernel into a clustered multikernel and (2) a list of theorems which have to be proved for the kernel in question. To assist with proving these theorems the framework provides a collection of proved kernel-agnostic theorems.

In this work, we apply the lifting framework to the seL4 microkernel only. Nonetheless, we claim that it can be applied to an arbitrary uniprocessor kernel under the following assumptions: (1) The kernel and its refinement proof need to be structured as outlined in Section 2 (distinct bootstrapping and runtime phases); and (2) the kernel needs to be event-based (no kernel-thread blocking allowed).

4.1 The seL4 Microkernel

As high-performance microkernel in the L4 family, seL4's security model supports fine-grained dissemination of authority via capabilities, which allows controlled communication between otherwise isolated components. In order to deal with concurrency, seL4 only supports uniprocessors, is event-based and non-preemptible with the exception of two well-defined preemption points.

The original seL4 verification project, L4.verified, has formally verified that the C implementation refines an intermediate executable specification written in Haskell, and that this specification in turn refines an abstract specification written in Isabelle/HOL. This transitively proves refinement between the implementation and the abstract specification. The verification framework is implemented in Isabelle/HOL and tailored to facilitate reasoning about sequential programs.

The initial implementation of seL4 was done for ARMv6, which is the version that is formally verified. The x86 version was initially ported from ARM directly on C level with its intermediate Haskell specification added later. There are plans to write an abstract specification and prove that it is refined by the implementation.

The clustered-multikernel implementation of seL4, *seL4::CMK*, is based on the x86 version of uniprocessor seL4. The main reason for this is the abundant availability of x86 multiprocessor/multicore systems compared to ARM, especially with a scalability evaluation in mind requiring NUMA and a high number of cores.

4.2 Turning seL4 into seL4::CMK

Applying the aforementioned recipe to seL4 resulted in the following changes to the implementation and the specification.

We prepended a first part of bootstrapping which handles starting CPUs: The very first CPU discovers devices, configures the platform, allocates memory to nodes, writes the configuration to a specific region of memory and starts the first CPU of each node. Within each node, the original uniprocessor bootstrapping version takes over, reads the configuration data and initialises the node's kernel data. It was modified to start the remaining CPUs of the node at the end. Further minor modifications were necessary to make access to static kernel data node-local and to change the way the configuration data is read at the beginning.

For the runtime phase, we modified the code around syscall entry/exit in order to implement the big lock. Furthermore, we had to split the static kernel data into two parts: (1) CPU-local and (2) shared between CPUs of the same node. Most of the state ended up in the shared part, with the exception of the x86 Task State Segment (TSS). It contains the pointer to the currently running thread and therefore needs to be CPU-local.

The new first part of bootstrapping comprises 350 LOC. The second part of bootstrapping is the original uniprocessor bootstrapping with a few dozen LOC of modifications. For the runtime phase, only 40 LOC of assembly code had to be modified in order to implement a big-lock design. No changes on the C level were necessary. The splitting into shared and CPU-local state needed to be implemented completely on the assembly level.

Compared to the overall size of seL4 (8700 LOC), these modifications are relatively small.

4.3 Lifting the Bootstrapping Phase

4.3.1 Kernel Isolation Theorem

First, we prove that after bootstrapping, the nodes' kernels are and stay isolated from each other. We do this because for the runtime phase (Section 4.4), we want to be able to reason about each node in isolation.

The *kernel isolation theorem* ensures isolation between the nodes' kernel memory and consists of two parts. The first part talks about the bootstrapping phase. Translated to prose, it says: "After having bootstrapped, any memory designated to create kernel data structures in is partitioned between nodes". The second part talks about the runtime phase: "Kernel data structures are never created outside the designated memory region."

4.3.2 Kernel-Memory-Access Theorem

The *kernel-memory-access theorem* shows that kernel bootstrapping behaves correctly with regards to concurrency by stating that every CPU observes sequential semantics locally, which is what the abstract specification relies on. The theorem is formulated on the level of memory-access histories: "For each CPU and memory location, between any read or write followed by a read any time later, no write by another CPU occurs."

As you remember, correspondence of the initialisation function (which covers both bootstrapping parts) is proved with the original verification framework. Previous work [19] presented how to augment that framework to support reasoning about concurrency. It involves generating a sequence of high-level instructions (memory read/write, memory fence, remote CPU start) from the abstract specification which is fed into an operational model of a multiprocessor architecture. This model computes all possible interleavings of memory accesses, taking into account the resulting start sequence of the CPUs. We have since extended this model to also cover TSO multiprocessor architectures, with exhibit weak memory ordering (Section 4.3.3).

This is necessary because the first part of bootstrapping is inherently concurrent. The first CPU needs to store some global configuration in a memory location where it will be read concurrently by the CPUs of the other nodes which it had just started.

Proving the *kernel-memory-access theorem* on the abstract level is permissible because memory accesses are overapproximated. This means that instead of directly modelling every assembly instruction that accesses memory (which is impossible on the abstract level), we specify that one or more accesses to a set of memory locations are performed for a specific abstract operation. The model then non-deterministically assumes all memory access patterns the implementation could possibly perform.

4.3.3 *Formal Model of Weak Memory Ordering*

The weak-memory model developed for this work is based on the Cambridge x86-TSO operational model [16] which it extends with support for CPUs starting other CPUs (also nested). Unlike the Cambridge model, it is a generic TSO (total store order) model and not specific to x86.

The main feature of a TSO architecture is the *store buffer*, which is also the sole source of memory reordering on such an architecture. Whenever an instruction executed by the CPU retires and triggers a memory write, the value is first stored in a FIFO buffer—the store buffer—which delays writing it to the memory subsystem[2] by an unspecified amount of time. On the other hand, reading is performed directly from the memory subsystem. Most TSO architectures support *store-buffer forwarding*, which enables a read directly from the store buffer in case the same memory address had been written to shortly before and the value is still in the store buffer. The TSO model presented here (and depicted in Figure 2) includes store-buffer forwarding and memory fences.

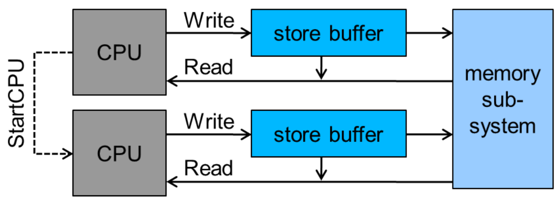

Figure 2: TSO Multiprocessor Architecture Model

Its formal specification in Isabelle/HOL requires 100 LOC. The proof of the kernel-agnostic *sequential-semantics theorem* (which is needed to prove the kernel-specific *kernel-memory-access theorem* presented above) encompasses 1000 LOC in Isabelle/HOL.

The novelty of this model is the combination of handling nested starting of CPUs combined with memory reordering. For example, it models the possibility of a memory write being reordered with the signal to start another CPU. While this behaviour has never been observed on x86 CPUs (and programmers often rely on it not to happen), neither the Intel nor the AMD architecture manuals state that this cannot happen. Notwithstanding, other TSO architectures are free to explicitly allow it.

For the kernel bootstrapping this means that we have to prove that an instruction starting another CPU is always preceded by a memory fence. Otherwise it is possible that the started CPU reads stale configuration data written by the CPU it has been started from.

4.3.4 *Application to seL4*

Verification of seL4's bootstrapping code was not in the scope of L4.verified. Hence, there was no existing abstract specification so we wrote the clustered-multikernel bootstrapping specification from scratch (1300 LOC). Proving the *kernel isolation theorem* and the *kernel-memory-access theorem* required 6400 LOC.

[2]The memory subsystem includes the memory itself and all caches layered on top of it. On TSO architectures, cache coherency protocols ensure sequential consistency of the memory subsystem.

The primary sources of proof complexity turned out to come from: (1) the fact that bootstrapping of a clustered multikernel requires nested CPU starts. (2) The configuration data is provided to the other CPUs via a specific region of memory. Because this region is written and then read concurrently, the proof needs to take into account memory fences and a correct CPU starting sequence. (3) During bootstrapping, seL4 uses dynamic memory allocation, which had a major impact on the bootstrapping-part of the *kernel isolation theorem*. Interestingly, its runtime counterpart happened to be a specialisation of an invariant already proved by L4.verified and could therefore be proved with a few dozen LOC.

In the meantime, an intermediate Haskell specification of the bootstrapping has been written for the uniprocessor version of seL4 and a correspondence proof is currently being worked on.

4.4 Lifting the Runtime Phase

For the runtime phase, the *kernel isolation theorem* allows us to reason about each node in isolation.

In a refinement proof, the refinement automaton models the actions of an execution unit, i.e. a CPU. Within a node of a clustered multikernel, we have multiple CPUs running in parallel, which requires us to model each node as a *parallel composition* of the uniprocessor refinement automaton.

As you remember from Section 4.2, a node's kernel state is divided up into a shared and a CPU-local part with the latter containing the "currently running thread" variable. Hence, we cannot just copy the uniprocessor refinement automaton and "run" the copies in parallel. Whenever one of the parallel automata makes a transition, it potentially modifies its own CPU-local state and the shared state but it cannot modify another CPU's state. This may sound harmless at first sight, but a closer look at it reveals that from another CPU's point of view, the shared state gets changed magically under its feet, something that could not happen to the state in the uniprocessor world.

To this end, this work includes a formal generic *automaton lifting operation* with accompanying *automaton lifting theorem*. The former lifts an arbitrary refinement automaton into a parallel composition of itself with a parameterisable splitting of the original state into shared and CPU-local parts. Each transition in the new parallel automaton consists of merging the CPU-local and shared states into the original state, executing the original transition and splitting the resulting state back into the new shared and CPU-local states. The transitions are interleaved non-deterministically. The interleaving model is applicable because we assume that the implementation puts a big lock around the entire kernel transition.

The *automaton lifting theorem* says: "When applying the lifting operation to the refinement automata of both abstract and concrete levels of an arbitrary refinement proof, the concrete parallel refinement automaton refines the abstract parallel refinement automaton if the original concrete refinement automaton refines the original abstract refinement automaton." The theorem can only be applied if certain properties about the lifting parameters hold. Most of them are trivial or obvious, e.g. that the splitting function splits the entire state, not just parts of it. Nonetheless, there is one not so obvious but all the more critical assumption: The original invariants need to be *splittable*. What does this mean?

Remember that correspondence proofs generally require an additional proof that certain invariants hold throughout execution. The problem now arises that splitting and independent modification of the shared and CPU-local parts potentially violates the invariants. In prose, the *splittable invariants* property is: "The original invariants do not talk about the shared and the CPU-local state at the same time", or in other words, "relate them to each other in any way". This property needs to be true (and proved) before the automaton can be automatically lifted.

The automaton lifting operation and theorem presented here are—to the best of our knowledge—the first ones to automatically lift an arbitrary refinement automaton (and its proof) into a parallel composition of itself with parameterisable splitting of the original state into shared and local parts. Specification and proofs in Isabelle/HOL comprise 150 LOC.

4.4.1 Application to seL4

The lifting theorem requires certain properties about the lifting parameters to hold. Most of them are trivial and could be proved with a few LOC. In contrast, the *splittable invariants* property turned out to be nasty to prove. More than that, it does not even hold for seL4: There are a handful of invariants which relate the shared with the CPU-local part of the state. Specifically, invariants talking about the currently running thread: The pointer to the currently running thread is in the CPU-local state whereas the thread-control block (TCB) is in the shared state. This means that we had to derive a more generic version of the lifting theorem which provides the possibility of dividing up the invariants into splittable and unsplittable ones. The splittable ones are handled automatically whereas the unsplittable ones have to be proved manually in the context of the new parallel refinement automaton.

In seL4, the unsplittable invariants ensure the valid state of the currently running thread. Trying to prove these invariants directly over the new parallel refinement automaton revealed an unexpected problem with the C implementation, even though no C code was changed: The *thread-deletion problem* is seL4-specific and a good example in showing the bug-finding abilities of theorem proving in general, and our lifting framework in particular. 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. On a uniprocessor, thread A knows that any other thread it is modifying/deleting is not currently running on a CPU. In seL4::CMK however, thread B could be running on another CPU. 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 state of thread B is not valid when it is deleted by thread A while running on another CPU because the current-thread pointer in that CPU's local state is now dangling.

Adding the necessary coordination to seL4::CMK to fix the problem required only 80 LOC of C code and a similar amount on the abstract level. This sounds relatively harmless. Nevertheless, due to the event-based structure of seL4 (specifically: no kernel-thread blocking allowed), two thread states and 8 preemption points had to be added. This increased proof complexity considerably. While fixing up the invariant proofs over the abstract specification was a moderate effort (100 LOC modified, 300 LOC added), adding a newly required invariant required a 1000-LOC proof.

Left for future work are the necessary changes to the intermediate Haskell specification and fixing up the correspondence proofs after those changes.

5. RELATED WORK

The clustered multikernel is an extension of the multikernel, which is the design introduced with Barrelfish [1] and also chosen by Corey [3]. Barrelfish is an OS aimed to be highly scalable and suitable for heterogeneous multiprocessing. It follows a distributed-system approach by keeping kernel data structures local to a CPU or replicated on other CPUs. Synchronisation and coordination between CPUs is message-based and managed by the system software running on top. The API of the underlying microkernel is inspired by seL4. Corey is an OS with almost the same aims as Barrelfish. Kernel data is CPU-local too, but system software is allowed to choose which kernel data should be shared between CPUs.

Clustered OS kernels emerged in the early 90s. Hurricane [18] used clustering to improve data locality on large-scale NUMA multiprocessors while Hive [5] aimed at fault isolation between clusters. While performing well for certain kinds of applications, these kernels suffered from high complexity and unpredictable performance. In the late 90s, Disco [4] and Tornado [9] tried to overcome these problems, albeit with very different approaches. Disco aimed at reducing implementation cost and complexity by reviving the idea of virtualisation, which would allow reusing general-purpose OSes on large-scale NUMA multiprocessors. Tornado aimed at better scalability by further improving data locality. The approach was to construct the kernel in an object-oriented manner which encapsulates related kernel data. The novel idea of *clustered objects* allowed kernel data locality to be fine-tuned.

On the formal side, Microsoft's VCC verification environment allows to reason about concurrent system-level C code. Proofs are guided by creating C annotations such that the generated proof obligations can be discharged automatically. Thread-local data can be reasoned about in a sequential context using an ownership discipline while concurrent data structures (e.g. locks) are handled separately. The Verisoft XT project used VCC to formally verify substantial parts of Microsoft's Hyper-V multiprocessor hypervisor [6]. The proved properties are mainly function contracts and invariants on data types. Unfortunately, these results are not sufficient to conclude an overall functional-correctness theorem from. Moreover, the VCC specification language is very close to C, which leads to low-level specifications that make it hard to prove higher-level properties such as isolation or integrity.

6. CONCLUSION

The clustered multikernel fulfils its promise as a multiprocessor design capable of turning a uniprocessor kernel into a multiprocessor version with minor modifications while only introducing a small amount of concurrency. The lifting framework is practical and can be applied to seL4, a uniprocessor kernel verified in a large-scale project.

We are not aware of a successful refinement proof of a multiprocessor kernel. Given a verified uniprocessor kernel, the clustered multikernel offers a way to achieve this with relatively low effort. In case of seL4::CMK, the entire proof effort (including the estimated LOC for the missing correspondence proofs) is around 12 kLOC. Compared to L4.verified's overall proof size of 200 kLOC [12], this is relatively small.

6.1 Future Work

6.1.1 Correspondence of C Code

The lifting framework concentrates on the abstract level and relies refinement to transfer the necessary theorems down to C level, which gives us the assurance that the implementation of the clustered multikernel corresponds to its specification. Hence, correspondence proofs need to be fixed if the code in question was modified. For seL4::CMK, two correspondence proofs are missing.

The first one is the correspondence proof of kernel bootstrapping. This proof was also missing in seL4's uniprocessor version but is now being worked on. After its completion, it can be ported to seL4::CMK. This requires some fixes due to the slightly modified second part of bootstrapping. The first part requires an additional correspondence proof.

The second missing correspondence proof belongs to the solution of the thread-deletion problem from Section 4.4.1. After its completion, the final refinement theorem can be extended to reach down to the C level.

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.

6.1.2 *Performance Evaluation*

On the systems side, the impact of the clustered multikernel's limitations is still unclear. While there is already plenty of supporting work for the multikernel [1], no conclusive evaluation exists on how far and under which workloads exactly a big-lock kernel scales/performs. Experience with early Linux lets us expect scalability problems. However, this was before the advent of multicore systems with their tight coupling between cores. Furthermore, if the kernel is sufficiently small (e.g. a microkernel or hypervisor) it is likely that the big lock scales up to a reasonable number of CPUs. An example that supports this conjecture is the OKL4 Microvisor [15], a small commercial microkernel/hypervisor implemented in the big-lock design: It has been successfully deployed in over 1.5 billion multicore mobile phones to date. Preliminary benchmarks of OKL4 on a NaviEngine platform (4 ARM11 MPCore CPUs) revealed that scalability was degraded more by cache contention than the big lock itself [13]. In other words: Fine-grained locks would not remove the scalability problem.

Looking at clustering, research from the early 90s does not look very promising (Section 5. However, these performance evaluations are based on large-scale NUMA machines of that time. Today's machines with their tight coupling of cores and large, hierarchical caches require different trade-offs. Furthermore, we conjecture that the high complexity and unpredictable performance of these systems are introduced by trying to hide clustering from the application and provide a single-system image. We suspect that this is not necessary for most of the applications you would run on a clustered multikernel today.

Consequently, the next step in this work is a thorough performance and scalability evaluation of the clustered-multikernel design based on benchmarks of seL4::CMK. The evaluation should answer the following question.

From a system designer's point of view, a multiprocessor kernel using traditional synchronisation primitives (such as fine-grained locks, lock-free) gives us (1) good scalability and (2) flexible kernel-resource usage across CPUs *at the same time*. For verification reasons, we restrict ourselves to a clustered multikernel where we need to trade one for the other. So the question is: For "any interesting" multiprocessor workload, can we cluster into nodes only the parts requiring flexible kernel-resource usage *and* (re)write the rest of the workload to run as a distributed system of nodes *and* still perform/scale?

7. REFERENCES

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *22nd SOSP*, Big Sky, MT, USA, Oct 2009. ACM.

[2] W. R. Bevier. Kit: A study in operating system verification. *Trans. Softw. Engin.*, 15(11):1382–1396, 1989.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *8th OSDI*, San Diego, CA, USA, Dec 2008.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *16th SOSP*, St. Malo, France, Oct 1997.

[5] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *15th SOSP*, Copper Mountain, CO, USA, Dec 1995.

[6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer-Verlag.

[7] M. Daum, J. Dörrenbächer, and B. Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *JAR: Special Issue Operat. Syst. Verification*, 42(2–4):349–388, 2009.

[8] M. Daum, N. W. Schirmer, and M. Schmidt. Implementation correctness of a real-time operating system. In *Int. Conf. Softw. Engin. & Formal Methods*, pages 23–32, Hanoi, Vietnam, 2009. Comp. Soc.

[9] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *3rd OSDI*, pages 87–100, New Orleans, LA, USA, Feb 1999.

[10] L. Hatton. Re-examining the fault density - component size connection. *Softw.*, 14(2):89–97, 1997.

[11] G. Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[13] A. Lyons. Efficient concurrency control for high-performance microkernels. BSc thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Jul 2011. Available from publications page at http://ssrg.nicta.com.au/.

[14] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[15] OK Labs: OKL4 Microvisor. http://www.ok-labs.com/products/okl4-microvisor.

[16] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7):89–97, Jul 2010.

[17] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA WS Operat. Syst. Verification*, Sydney, Australia, Oct 2004.

[18] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *J. Supercomp.*, 9:105–134, 1993.

[19] M. von Tessin. Towards high-assurance multiprocessor virtualisation. In *6th VERIFY*, Edinburgh, UK, Jul 2010.

[20] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.