Reasoning about Concurrency in High-Assurance, High-Performance Software Systems

June Andronick

Data61, CSIRO (formerly NICTA) and UNSW, Sydney, Australia
june.andronick@data61.csiro.au

Abstract. We describe our work in the Trustworthy Systems group at Data61 (formerly NICTA) in reasoning about concurrency in high-assurance, high-performance software systems, in which concurrency may come from three different sources: multiple cores, interrupts and application-level interleaving.

Formal verification – mentality shift

Recent years have seen a shift in the perception of formal software verification in the academic community and, to some more emerging extent, in the industrial community. The strength of a mathematical proof to guarantee the correctness, security and safety of programs deployed in high-assurance systems has made its way from utopia to reality, and the absence of such strong evidence will hopefully soon be considered negligence for critical systems.

This shift was possible thanks to highly successful verified artifacts, such as the CompCert compiler [16] and the seL4 operating system (OS) kernel [14,15]. A remaining grand challenge in formal software verification is *concurrency reasoning*, much harder than sequential reasoning because of the explosion of the number of interleaved executions that need to be considered.

Concurrency in software systems can have three different sources: multiple cores, interrupts and application-level interleaving. In this paper we first briefly explain these kinds of concurrency and their challenges, and we then describe our recent and current work in providing concurrency reasoning framework and verifying concurrent software systems in these three areas.

Software systems and concurrency – background

Multicore platforms provide a computing power boost that is hard to resist for a very competitive software system market, even for high-security solutions. Code execution can be parallelised on different cores, and the challenge, for implementation as well as verification, is to ensure safe sharing between cores. This can be done using locking mechanisms: a core can access shared data only after acquiring a lock guaranteeing that no other core is manipulating the data at the same time. This effectively eliminates concurrency, but has a performance

impact and bears liveness risks (e.g. potential for deadlocks). Another option is to let cores access shared data without any locking, relying on more indirect arguments that the resulting race conditions are still safe. This requires much more careful reasoning.

Interrupts introduce a different kind of concurrency, where interleaving is *controlled* in the sense that the only thing that truly happens in parallel with code execution is the occurrence of interrupts (i.e. a flag's being set in hardware). The code being executed can still be stopped at any time, and control switched to handler code that will service the interrupt; but the execution of the handler code is then sequential until the return from interrupt (except when nested interrupts are supported, in which case further interleaving is allowed). Handler code and "normal" code may share data (e.g. the list of runnable threads), whose access needs to be carefully designed. Once again, there is a radical way of ensuring safe sharing: manually switching off interrupts during manipulation of data shared with handlers. However, that has a performance and latency impact.

Application-level, or user-level, concurrency is another form of controlled concurrency. In an OS-based system, the OS kernel provides hardware abstraction primitives, such as threads, to applications. Threads run concurrently in the sense that the OS kernel will simulate parallel execution through scheduling and time sharing between threads. For better latency, threads are often preemptible by the kernel: their execution can be paused at any time by the kernel, their execution context saved, and execution switched to another thread. Safe memory sharing between threads can also be handled via locking mechanisms, where more feature-rich sychronisation mechanisms can be provided by the kernel.

For all these types of concurrency, the general trend on the reasoning and verification side is to aim for limiting the concurrency as much as possible: local operations can be parallelised, but sharing should be done only when mutual exclusion can be guaranteed (by locking or other indirect arguments). This approach is the basis of many existing verification frameworks and verified systems (e.g. [18,8,9,11]).

However, on the implementation side, the trend goes for more racing to improve performance: some systems need to run with interrupts enabled as much as possible, or to run some critical code unlocked. We are targeting such real-world systems, where the possible races need to be proven not to violate the desired properties for the system.

Interrupt-induced concurrency

Our work on reasoning about interrupt-induced concurrency is initially motivated by the verification of eChronos [2], a small embedded real-time operating system in commercial use in medical devices. In an eChronos-based system, the kernel runs with interrupts enabled, even during scheduling operations, to be able to satisfy stringent latency requirements. The formal verification of eChronos' correctness and key properties thus required a reasoning framework for controlled concurrency that describes interleaving between "normal" code (application code

and kernel code) and interrupt-handler code. We want such a framework to support potentially racy sharing between handlers and normal code, rather than having to bear the cost of interrupt disabling to ensure safe sharing.

We developed a simple, yet scalable framework for such controlled interleaving and have used it to define a high-level model of eChronos scheduling behavior [7]. We then proved its main scheduling property: that the running task is always the highest-priority runnable task [6]. Our framework is embedded in Isabelle/HOL [17] and the verification relies on the automation of modern theorem provers to automatically discharge most of the generated proof obligations. Our models and proofs are available online [1].

Our modelling framework builds on foundational methods for fine-grained concurrency, with support for explicit concurrency control and the composition of multiple, independently proven invariants. The foundational method is Owicki-Gries [19], a simple extension on Hoare logic with parallel composition, *await* statements for synchronisation, and rules to reason about such programs by inserting assertions, proving their (local) correctness sequentially as in Hoare logic, and then proving that they are not interfered with by any other statement in parallel.

We model an interruptible software system as a parallel composition of its code with code from a number of interrupt handlers. We also model the hardware mechanisms that switch execution to handlers and that return from interrupts, via the scheduler. Such parallel composition allows more interleaving than can happen in reality – for instance it allows the execution of the handler code suddenly to jump back to executing application code at any time. We therefore then restrict the interleaving by a control mechanism, that we call *await painting*: every instruction is guarded by a condition, which by default enforces sequential execution, but is relaxed for all hardware mechanisms that do allow interleaving, such as taking an interrupt or returning from one.

For the verification, the main property of interest in an invariant, which, as most invariants, rely on a number of helper invariants. To make the verification scalable, we have a compositionality theorem allowing the proof of helper lemmas independently, with separate Owicki-Gries assertions, after which those invariants can be assumed when proving further invariants. We have also developed proof-engineering techniques to address scalability issues in the verification of the generated proof obligations. These techniques range from subgoal deduplicating and caching, to exploiting Isabelle's parallelisation and powerful simplifier.

With this framework, we proved eChronos' main scheduling property with a single tactic application. This proof is about a high-level model of eChronos and the obvious missing piece is the link to the implementation.

To bridge the gap to the implementation, we have developed a verification framework for concurrent C-like programs, called COMPLX [3], available online [10]. The COMPLX language builds on SIMPL [22], a generic imperative, sequential language embedded in Isabelle/HOL. SIMPL allows formal reasoning about sequential C programs via the translation of C programs into SIMPL by the C-to-Isabelle translation [24]. It has been used for the verification of seL4: the

C-level formal specification of seL4 is in SIMPL, inside Isabelle/HOL. COMPLX extends SIMPL with parallel composition and await statements, and we developed a logic for Owicki-Gries reasoning as well as its compositional counter-part Rely-Guarantee reasoning [13]. Using this framework to extend the eChronos verification to the implementation and to full functional correctness is future work. We are also planning to use it in our ongoing verification of the multicore version of seL4.

Multicore concurrency

The seL4 microkernel is a landmark in software verification [14,15]. It is the world's "most verified" OS kernel, while also being the world's fastest operating system designed for security/safety. It has formal, mechanically checked theorems for functional correctness, binary verification, integrity- and information-flow security, and verified system initialisation. It has seen 3rd-party use, demonstrated in automotive, aviation, space, military, data distribution, IoT, component OS, and military/intelligence . It is also the only verified kernel that has been maintained, extended with new features and ported to new platforms over a number of years. A direct implication is a very large (and evolving) proof stack (0.74M lines of specifications and proofs). One of the remaining challenges is to extend the formal verification, so far for unicore platforms, to multicore.

A multicore version of seL4 has been developed following a (mostly – as we will explain shortly) big-lock kernel approach. The idea of a big-lock kernel allows us to run kernel-based systems on multicore machines, where the user code can make use of the multicore computation power, while parallelism during kernel calls is reduced by a so called "big lock" around all kernel executions. Recent work in our group [20] indicates that this coarse-grained locking approach, at least for a well-designed microkernel with short system calls, can have less overhead than a fine-grained locking approach on modern hardware, and performs indistinguishably from fine-grained locking in macro-benchmarks on processors with up to 8 cores. The reason is that the time spent inside a fast microkernel using big lock is comparable to the time spent in fine-grained locks in monolithic kernels like Linux. Fine-grained locking is traditionally used for scalable multicore implementations, but comes with considerable complexity. Since the big-lock approach implies a drastic reduction in interleaving, it makes real-world verification of multicore kernels feasible.

The challenges in verifying this multicore seL4 are manifold. Firstly, the kernel is only *mostly* locked when executed. Some kernel code executes outside of the lock, for performance reasons and to deal with unavoidable hardware-software sharing. Indeed some hardware registers that are shared between cores are accessed by critical code in kernel calls, such as the deletion of a thread from another core. These operations cannot be locked and need careful design and reasoning to avoid data corruption. To start addressing this, we have performed a formal proof that the validity of such critical registers is always preserved. This involves proving the correctness of the complex OS design for deletion on multicore. This proof is done on a very high-level model of interleaving (reusing

the verification framework from our eChronos verification). It still needs to be connected to more concrete models of seL4, but it already identifies the guarantees that need to be provided by each core for the safe execution of the other cores.

The second challenge is to identify correctly the shared state between cores. This can be shared state between user code on one core and kernel code on another core, or shared state between two instances of kernel execution (at least one unlocked). There exists an earlier formal argument for an experimental multicore version of seL4 that lifts large parts of the sequential functional correctness proof to the multicore version [23]. This version relies on an informal identification of the shared state between the kernel and the user components (and very limited code outside the lock), and an informal argument that this shared state does not interfere (and therefore cannot invalidate) the kernel's (sequential) correctness result.

In our current work, we are aiming for a more foundational verification, and support for kernel-to-kernel interaction. We want to model all possible interference, then exclude the impossible ones *by proof* and finally show that the remaining ones do not violate the kernel invariants and properties. In particular we want, at the bottom level, to model explicitly the parallel composition of cores, with a framework like COMPLX (with potentially further work to port the guarantees to binary and to weak memory). This raises the question of bridging the gap, through refinement, between the high-level model of multicore seL4, a functional specification of seL4 and the lowest implementation level.

This leads to the remaining challenge, which is to leverage the existing large proof stack, whose complexity reflects the complexity of a high-performance, non-modular microkernel. This is ongoing work. We are aiming for an approach that will preserve as much as possible the sequential specifications and the corresponding refinement theorems.

User-level concurrency

Our vision for proving security for entire large systems [4,12,21] is to build them on a trustworthy foundation like seL4 and then to leverage its isolation properties in a way that the applications can be componentised into trusted- and untrusted components, avoiding in particular having to verify any of the untrusted components, thanks to the kernel's integrity and confidentiality enforcement.

We have previously built [5] an initial prototype framework that provides, for such microkernel-based, componentised systems, and for any targeted system invariant, a list of proof obligations. Once proved by the user of the framework, these theorems will imply that the invariant is preserved at the source code level of the whole system. We have already demonstrated this approach on a simplistic system with two components: a small trusted component with write access to a critical memory area, and one potentially very large untrusted component with only read access to the same region and otherwise isolated. We were able to prove properties about the memory content without any proof about the untrusted components, relying only on seL4's integrity enforcement.

This approach however suffered from strong limitations in terms of scalability and the kind of properties supported (they needed to rely solely on integrity enforcement). This piece of work was prior to the more foundational treatment of concurrency we developed more recently for the ongoing verification of eChronos and multicore seL4. Our current aim is to incorporate the possibility of user-level reasoning in the modelling and refinement framework currently developed for the multicore seL4, with proper explicit modelling of user-to-user interactions and the specification of rely- and guarantee conditions.

Tackling the formal verification of concurrent high-performance software systems is both challenging due to the combined complexity of high-performance and concurrency, and indispensable to keep such systems real-world relevant. We have presented challenges, progress made, and future work in building reasoning frameworks that can support such scale and complexity, and their application to the verification of real-world operating systems such as eChronos and seL4.

Acknowledgements.
The author would like to thank the people that have worked on the research presented in this paper: Sidney Amani, Maksym Bortin, Gerwin Klein, Corey Lewis, Daniel Matichuk, Carroll Morgan, Christine Rizkallah , and Joseph Tuong. The author also thanks Carroll Morgan, Gerwin Klein and Gernot Heiser for their feedback on drafts of this paper.
Parts of the work presented are supported by the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AOARD) and U.S. Army International Technology Center - Pacific under grant FA2386-15-1-4055. Other parts have been supported by AOARD grants FA2386-12-1-4022 and FA2386-10-1-4105.

References

1. eChronos model and proofs, `https://github.com/echronos/echronos-proofs`
2. The eChronos OS, `http://echronos.systems`
3. Amani, S., Andronick, J., Bortin, M., Lewis, C., Christine, R., Tuong, J.: Complx: A verification framework for concurrent imperative programs. In: Yves Bertot and Viktor Vafeiadis (ed.) CPP. pp. 138–150. ACM, Paris, France (Jan 2017)
4. Andronick, J., Greenaway, D., Elphinstone, K.: Towards proving security in the presence of large untrusted components. In: Ralf Huuck, Gerwin Klein, Bastian Schlich (ed.) SSV. p. 9. USENIX, Vancouver, Canada (Oct 2010)
5. Andronick, J., Klein, G.: Formal system verification - extension 2, final report AOARD #FA2386-12-1-4022. Technical report, NICTA, Sydney, Australia (Aug 2012)
6. Andronick, J., Lewis, C., Matichuk, D., Morgan, C., Rizkallah, C.: Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. In: Jasmin Christian Blanchette and Stephan Merz (ed.) ITP. pp. 52–68. Springer, Nancy, France (Aug 2016)

7. Andronick, J., Lewis, C., Morgan, C.: Controlled owicki-gries concurrency: Reasoning about the preemptible eChronos embedded operating system. In: Rob J. van Glabbeek and Jan Friso Groote and Peter Höfner (ed.) Workshop on Models for Formal Analysis of Real Systems (MARS 2015). pp. 10–24. Suva, Fiji (Nov 2015)
8. Appel, A.: Verified software toolchain. In: Barthe, G. (ed.) 20th ESOP. LNCS, vol. 6602, pp. 1–17. Springer (2011)
9. Chen, H., Wu, X.N., Shao, Z., Lockerman, J., Gu, R.: Toward compositional verification of interruptible os kernels and device drivers. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 431–447. PLDI '16, ACM, New York, NY, USA (2016)
10. COMPLX entry in the Archive of Formal Proofs. `https://www.isa-afp.org/entries/Complx.shtml`
11. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: OSDI (Nov 2016)
12. Heiser, G., Andronick, J., Elphinstone, K., Klein, G., Kuz, I., Ryzhyk, L.: The road to trustworthy systems. In: ACMSTC. pp. 3–10. ACM, Chicago, IL, USA (Oct 2010)
13. Jones, C.B.: Tentative steps towards a development method for interfering programs. Trans. Progr. Lang. & Syst. 5(4), 596–619 (1983)
14. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. CACM 53(6), 107–115 (Jun 2010)
15. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. Trans. Comp. Syst. 32(1), 2:1–2:70 (Feb 2014)
16. Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) 33rd POPL. pp. 42–54. ACM, Charleston, SC, USA (2006)
17. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
18. OHearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (Apr 2007)
19. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica 6, 319–340 (1976)
20. Peters, S., Danis, A., Elphinstone, K., Heiser, G.: For a microkernel, a big lock is fine. In: APSys. Tokyo, JP (Jul 2015)
21. Potts, D., Bourquin, R., Andresen, L., Andronick, J., Klein, G., Heiser, G.: Mathematically verified software kernels: Raising the bar for high assurance implementations. Technical report, NICTA, Sydney, Australia (Jul 2014)
22. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
23. von Tessin, M.: The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels. PhD thesis, School Comp. Sci. & Engin., UNSW, Sydney, Australia, Sydney, Australia (Dec 2013)
24. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Martin Hofmann and Matthias Felleisen (ed.) POPL. pp. 97–108. ACM, Nice, France (Jan 2007)