Resurrecting seL4's WCET Analysis

Simon Sillitoe[1], Abdul Basit[2], Massimiliano Giacometti[2], Gernot Heiser[1]
[1]UNSW Sydney, Australia
[2]PlanV Tech, Munich, Germany
{s.sillitoe,gernot}@unsw.edu.au, {abdul.basit,massimiliano.giacometti}@planv.tech

***Abstract*—We report on our ongoing work to re-establish the WCET analysis of the latest version of the verified seL4 microkernel on the 64-bit RISC-V architecture. By modifying the Heptane analysis tool to include missing features and remove scalability problems, we can now perform the analysis on an unmodified kernel, currently using place-holder latencies and loop bounds. Establishing high-assurance loop bounds and infeasible-path refutations is in progress, as is determining sound instruction latencies.**

1. Original WCET Analysis of seL4

Fifteen years ago, Blackham et al. performed the first sound and complete worst-case execution time (WCET) analysis of a protected-mode operating system (OS) kernel [2]. They performed the analysis on the Armv7 (32-bit) version of the seL4 microkernel, whose C implementation had been proved functionally correct [6]. That verification made seL4 a high-assurance foundation for safety- and security-critical systems, and thus an apt target for WCET analysis. Later, Sewell et al. strengthened the analysis by proving loop bounds and infeasible-path refutations on the source-code level, and then using a translation-validation toolchain to pass these into the WCET analysis [16].

This earlier work was based on the Chronos tool [7], albeit heavily modified to deal with the complexities of the kernel's 10 k source lines of code (SLOC) code base. However, Chronos became unmaintained even before that line of work was finished. Furthermore, with the introduction of out-of-order (OoO) cores, Arm discontinued publishing instruction latencies even for their in-order (IO) cores,[1] making it infeasible to maintain seL4's WCET analysis. Consequently, seL4 lost any hard real-time guarantees.

2. seL4 Developments

seL4 has undergone significant development since this early work, which not only means that the original WCET analysis has become completely meaningless, but also that re-establishing it has become non-trivial.

One major development is the move to 64-bit processors, especially Armv8 and RV64 (the 64-bit variant of RISC-V), including re-verification [15]. While the larger word size carries the risk of introducing scaling issues in static-analysis tools, support for the open RISC-V architecture with readily available open-source processor implementations [11], opens new avenues for obtaining latency information.

The second major change was the introduction of the *mixed-criticality systems* (MCS) variant, which (among other things) elevates time to a first-class resource with capability-authorised budgets [8]; the MCS variant has just been verified [14].

3. Resurrecting the WCET Analysis

We recently embarked on an effort to re-establish seL4's WCET analysis. Our past experience made it clear that we needed an open-source analysis tool to be able to deal with the challenges posed by a real-world OS kernel. WCET analysis is generally performed on relatively simple control code, as well as benchmark suites that are also fairly simple [4].

We selected Heptane [5] as the most promising candidate. Alternatives include OTAWA [1], which seems unmaintained,[2] and Platin [9], whose integration with LLVM/clang is a disincentive for us.

As expected, Heptane required some effort to get it to the point where it was able to process the complete seL4 kernel. The specific areas requiring work were:

- Heptane's support for the RISC-V architecture was incomplete. While sufficient for analysing application code, it lacked support for the instructions accessing the control and status registers (CSRs) as specified in the "Zicsr" extension to the base architecture [13], certain other supervisor-mode instructions, and a number of arithmetic and shift variants emitted by the compiler for kernel code. We added support for the missing instructions, the supervisor CSRs they reference, and the corresponding instruction formats. We also replaced the MIPS-derived RISC-V data-address simulation with a native implementation.
- The tool lacked support for control-flow patterns that are required by the kernel. For example, while usermode execution completes by returning from the

1. Arm later resumed publishing instruction latencies for their IO cores.

2. We received no response to our attempts to contact the OTAWA maintainers.

`main()` function, the kernel returns from a system call by executing a specific instruction sequence ending in `sret` which performs the mode switch back to the user. This must be treated as a terminal node in the control-flow graph (CFG) rather than a normal return. Separately, the kernel relies on non-local jumps that bypass the standard call/return structure. Its call graph also contains bounded recursion, requiring configurable recursion bounds in the context-tree expansion.

- Important compiler optimisations were not supported, such as tail and sibling calls, where function returns bypass the original caller.
- Heptane's data-cache analysis originally only tracked accesses on the stack and through global pointers, while seL4 commonly dereferences arbitrary pointers. We extended the underlying data-address analysis with a pointer-keyed memory map, which the cache analysis then consumes unchanged.
- Parts of the implementation were too simplistic to scale to the size and complexity of the seL4 code base. This includes the fixed-point iteration algorithm for the cache analyses, which, while theoretically sound, had high computational complexity. We achieved significant speed-up by re-engineering this algorithm to leverage an ancillary data structure of the fully inlined control-flow graph and to process nodes in a more efficient order, avoiding redundant recomputation at CFG merge points. Additionally, we removed redundant ILP variable declarations, fixed memory leaks by adopting RAII idioms, and avoided copying large amounts of analysis state.

In the end, while substantial changes were required (102 files changed, with 5,182 SLOC added and 4,918 SLOC removed), the required effort was less than we feared. The first author, a recent graduate with no prior experience with WCET analysis nor formal methods in general, spent about five to six months of full-time work on the project, to the point where the tool analyses the complete RV64 code base of seL4.

For just over 11,000 SLOC, the total run time is approximately 3 h (around 1 h of analysis and a further 2 h for ILP solving). This is consistent with the experience from our earlier work [2], which took >4h (on older hardware) for a somewhat simpler kernel of 8,700 SLOC.

4. Present State

We analyse the 64-bit RISC-V kernel built from the verified MCS configuration (`RISCV64_MCS_verified.cmake`) at seL4 commit `deec818`, with the compressed-instruction extension (RVC) disabled and the compiler directed not to emit jump tables (`-fno-jump-tables`), so that the binary is amenable to the analysis.

While we can now analyse the complete kernel, the numeric WCET results are currently fairly meaningless, as we have set it up with placeholder latencies and loop bounds. Specifically we assume 3-cycle latency for all non-memory instructions, 1-cycle L1-cache hit, 10-cycle L2 hit, 100-cycle L2 miss, and all loop bounds being 10. This results in a kernel WCET (i.e. worst-case latency from kernel entry for any reason until kernel exit) of about 21 M cycles. But, given the arbitrary latency assumptions, and (so far) no elimination of infeasible paths, this is no more than an order-of-magnitude estimate.

TABLE 1. PER-ENTRY WORST-CASE LATENCY AND LONGEST-PATH INSTRUCTION COUNT, UNDER THE PLACEHOLDER LATENCIES AND LOOP BOUNDS DESCRIBED IN THE TEXT.

Entry	WCET (k cycles)	Instructions (k)
System call	21,000	730
Interrupt	470	16
Exception	430	15
`Call` (fastpath)	19	0.66
`ReplyRecv` (fastpath)	27	1.05

Table 1 breaks this down by kernel entry, giving the worst-case latency and the number of instructions on the longest path for each. The overall kernel WCET is dominated by the (slowpath) system-call entry.

These figures should be read against the size of the problem. The static control-flow graph is modest: 154 functions, around 3.2 k basic blocks joined by 4.4 k edges, and some 13 k instructions, with 50 loops. The context-sensitive analysis, however, distinguishes roughly 110 k distinct calling contexts, giving 1.5 M contextual basic blocks, and the resulting integer linear program has about 3.1 M variables and 5.9 M constraints. This growth is what motivates the algorithmic changes needed to scale the analysis to the full kernel.

5. Ongoing Work

We are working towards establishing sound parameters to make the analysis result meaningful, as well as extracting more detailed WCET values.

5.1. Verified loop bounds and path refutations

We are working on re-establishing the transfer of loop bounds (and infeasible-path refutations) proved at the source-code level into the binary analysis, by resurrecting the work of Sewell et al. [16]. The toolchain now runs on the RISC-V kernel. A handful of loops walk runtime-sized kernel structures, such as capability lookup tables, message cancellation queues, and the priority-sorted scheduling-context queues introduced by the seL4 MCS variant. These have no static bound; they are instead constrained by system design, such as the kernel's preemption mechanism or the static system configuration, and will require a parameterised WCET or additional preemption points in the kernel.

5.2. Sound instruction-latency bounds

CVA6 [10] is a 64-bit RISC-V core we use in a number of projects. Its open-source nature allows us to establish sound instruction latencies using a two-step process.

We first obtain an empirical latency bound for each functional unit (FU) implemented in the execution stage of the CPU. We isolate the unit by manual inspection of the SystemVerilog source code and cross-checking against the CVA6 user manual and microarchitectural documentation. We then exercise the FU in isolation under constrained-random stimulus, using randomised assembler code targeting the individual FUs, with randomised operand patterns. We record as a *candidate bound* the maximum observed cycle distance between the input and the output valid signal, by instrumenting the SystemVerilog code. While we use Siemens Questa as simulator, any SystemVerilog-capable one would be suitable.

Obviously these empirical latencies are unsound, as simulation can only sample a subset of operand patterns and control-flow paths, and documentation can contain errors. However, simulation is an effective way to find a candidate value while avoiding pessimism.

We then encode each candidate latency as a bounded-delay SystemVerilog Assertion (SVA) and use open-source model checker SymbiYosys [17] to prove the assertion. If the proof succeeds, the candidate is confirmed as a sound upper bound to be used in Heptane. Else SymbiYosys provides a counterexample trace which refutes the claim. In this case we either tighten the harness assumptions or raise the bound to a value that can actually be proved.

This work is expected to be completed in Q1'27. Meanwhile we plan to extract approximate (unsound) latencies from simulations as an interim solution.

We also expect to perform the WCET analysis on a suitable IO Armv8 processor.

5.3. More detailed analysis and optimisation

Heptane supports context-sensitive WCET analysis, which allows us to obtain different latencies for different types of kernel entries. We have used this to distinguish between the high-level entry types, i.e. interrupts, exceptions and system calls, obtaining a separate WCET bound for each, as well as for the `Call` and `ReplyRecv` fastpaths (Table 1). Distinguishing between the individual slowpath system calls is ongoing work, and will enable even less pessimistic schedulability analysis.

Once we have actual latencies, we expect to spend some time on optimising kernel code to reduce the WCET, similar to our earlier work [3].

6. Conclusions

In summary, we have succeeded in re-establishing the WCET analysis of the latest (more complex) version of seL4 for 64-bit RISC-V. We found the Heptane tool a suitable starting point and could extend/improve it to the point where it could process the complete kernel. Our changes have been upstreamed to the public Heptane code repository [12].

Acknowledgements

This work is being performed under the PISTIs-V project, funded by the German agency *Agentur für Innovation in der Cybersicherheit GmbH* (Cyberagentur) under the *Ecosystem Formally Verifiable IT – Provable Cybersecurity* (EVIT) Program. We thank Isabelle Puaut for her help with using Heptane.

References

[1] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, Lecture Notes in Computer Science, pages 35–46. Springer, 2010.

[2] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011. IEEE Computer Society.

[3] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys Conference*, pages 323–336, Bern, Switzerland, April 2012. USENIX.

[4] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *Workshop on Worst-Case Execution-Time Analysis*, pages 137–147, Brussels, BE, July 2010. OCG.

[5] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In *Workshop on Worst-Case Execution-Time Analysis*, pages 1–8. Schloss Dagstuhl - Leibniz-Zentrum Informatik, 2017.

[6] 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 *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[7] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, 69(1–3): 56–67, December 2007.

[8] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

[9] Emad Jacob Maroun, Eva Dengler, Christian Dietrich, Stefan Hepp, Henriette Herzog, Benedikt Huber, Jens Knoop, Daniel Wiltsche-Prokesch, Peter Puschner,

Phillip Raffeck, Martin Schoeberl, Simon Schuster, and Peter Wägemann. The Platin multi-target worst-case analysis tool. In *Workshop on Worst-Case Execution-Time Analysis*, page 2:1–2:14, Lille, France, July 2024. Schloss Dagstuhl - Leibniz-Zentrum Informatik.

[10] Open Hardware Foundation. CVA6: An application class RISC-V CPU core, 2020. URL https://docs.openhwgroup.org/projects/cva6-user-manual/index.html. Accessed 2026-06-26.

[11] Open Hardware Foundation. Projects, 2026. URL https://openhwfoundation.org/projects/. Accessed 2026-06-26.

[12] Isabelle Puaut. Heptane. URL https://gitlab.inria.fr/puaut/heptane. Accessed 2026-06-26.

[13] RISC-V International. "Zics" extension for control and status register (CSR) instructions, 2026. URL https://docs.riscv.org/reference/isa/unpriv/zicsr.html. Accessed 2026-06-26.

[14] seL4 Foundation. Functional correctness proof for MCS seL4 completed on RISC-V, June 2026. URL https://sel4.systems/news/2026.html#mcs26.

[15] seL4 Foundation. Verified configurations, 2026. URL https://docs.sel4.systems/projects/sel4/verified-configurations.html. Accessed 2026-06-26.

[16] Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Systems*, 53:812–853, September 2017.

[17] YosysHQ GmbH. SymbiYosys (sby) documentation, 2023. URL https://symbiyosys.readthedocs.io/en/latest/. Accessed 2026-06-26.