Protected Hard Real-time: The Next Frontier

Bernard Blackham, Yao Shi and Gernot Heiser
NICTA and University of New South Wales
firstname.lastname@nicta.com.au

Abstract

Hard real-time systems are typically written to execute either on bare metal or on a small real-time executive that offers no memory protection. This model scales poorly as systems become more complex and integrated, as is the trend in industry today. Designing hard real-time systems on a protected OS is often avoided due to the difficulty in predicting its response time.

Hard real-time systems with full virtual memory and memory protection have been proposed previously. However, to our knowledge, no such system has determined safe upper bounds on the latency introduced by this protection.

This paper proposes that hard real-time systems can be constructed confidently and cost-effectively using an operating system providing full memory protection and virtual memory. We contend that a carefully written microkernel providing these mechanisms has the ability to be used in a hard real-time system without overly pessimistic response time guarantees. We use the seL4 microkernel as a case study, investigating how the features of seL4's design enable a highly accurate WCET analysis.

1 Introduction

Traditionally, hard real-time systems are constructed on hardware with predictable timing characteristics and with minimal software "glue" between the application and the hardware itself. Such systems are often developed without an operating system—on "bare metal"—or use a lightweight real-time executive to schedule threads. They offer no memory protection between components. The lack of fault tolerance leads to a design that is difficult to confidently scale to complicated systems which integrate several complex software stacks on one processor.

Large systems often separate out critical real-time functionality onto dedicated processors, such as the baseband processor found on most smart phones. However, as manufacturers strive to gain a competitive advantage by adding features to embedded devices, the level of integration will only increase. Using dedicated processors does not scale—for example, cars and aircraft are trending towards combining both critical and convenience functionality, and the cost and weight of tens or hundreds of processors is a serious issue.

An alternative solution to satisfy this growing trend is to consolidate these systems onto a single processor, and use an operating system to provide isolation between critical real-time components, and less critical time-sharing components [MHH02]. However, for hard real-time designs, this solution depends on the ability to provide safe upper bounds on the interrupt latency of the OS. In most systems, the interrupt latency is determined by the maximum worst-case execution time (WCET) of all non-preemptible code in the kernel.

It is possible to achieve very good interrupt latencies by making the kernel fully preemptible. In this model, interrupts are permitted to occur anywhere within the kernel, except within some small protected regions of code, usually to modify critical data structures. This gives typical interrupt latencies in the order of tens, or hundreds, of cycles. However, this requires very careful coding of the interrupt paths, and defensively analysing that at every point in the kernel an interrupt cannot cause a crash or make the kernel's state inconsistent. Analysing concurrency issues of this nature is extremely challenging due to the explosion of possible interleavings to consider and the difficulty in reproducing timing-related bugs. Much research effort has been devoted to developing methods and tools to identify such cases.

Many kernels do not allow interrupts to occur whilst executing kernel code, or allow them only to occur at designated preemption points. This greatly simplifies the design and testing of the kernel, at the cost of higher worst-case interrupt latency. Well-placed preemption points mitigate the issue, but still cannot achieve the small latencies of the fully preemptible model.

As embedded processors become faster, guaranteed latencies in the 100 000s of cycles become acceptable for many applications. This, in turn, permits the integration of larger, more complex software components into a single system, keeping device costs lower. The challenge then for hard real-time systems is to compute a safe upper bound of the interrupt latency of the kernel.

Safe upper bounds for WCET are generally computed using a combination of static analysis techniques and measurements on real hardware [KWRP05, PZH07]. Operating systems kernels have long been an elusive target of static WCET analysis, due to their unstructured code, tight coupling with hardware, and sheer size. WCET bounds based on measurements alone cannot be relied upon—for example, measurement-based upper bounds stated for RTLinux [YB97] were later shown to be invalid [MHSH01]. Several kernels have been analysed using static analysis, including RTEMS [CP01] and OSE [SEGL04, CEE+02], but these did not support memory protection using paged virtual memory. To our knowledge, no kernel providing full virtual memory and memory protection has been successfully analysed for its WCET. An analysis was attempted on the L4 Pistachio kernel [SP07], but safe WCET bounds were never established.

We assert that a well-designed microkernel lends itself to a tight WCET analysis. This paper focuses on the seL4 mi-

crokernel and presents it as a viable solution for creating hard real-time systems with very strong isolation properties.

1.1 seL4 as a Hard Real-time Platform

seL4 [KEH+09] is a third-generation microkernel, broadly based on the concepts of L4 [Lie96]. It provides virtual address spaces, threads, communication via synchronous and asynchronous IPC, and capabilities for managing authority. The distinguishing feature of seL4 is that it is the only kernel to date with a formal machine-checked proof that the C code implementation adheres to the specification of the kernel. This additionally ensures that seL4 will never crash or perform an unsafe operation. Whilst these strong functional guarantees are sufficient for many systems, critical real-time systems also require temporal guarantees to achieve safety.

seL4's specification dictates that the kernel will never enter an infinite loop—i.e., all seL4 system calls eventually return to the user. Previously, this was the only temporal guarantee known of seL4. In this paper, we investigate seL4's application to hard real-time domains and present the benefits of analysing a formally verified kernel. A microkernel with provably correct operation and guaranteed worst-case execution time bounds creates a foundation on which large-scale, trustworthy, hard real-time systems can be built.

1.2 Contribution

This paper asserts that it is possible to compute realistic safe upper bounds on interrupt latency for protected microkernel-based systems. We demonstrate the first full WCET analysis of a memory-protected OS kernel, seL4, with a view to tuning the kernel for hard real-time applications. We perform a full *context-aware* analysis of all of seL4's code paths—specifically, the analysis virtually inlines all functions within seL4 so that it is context-sensitive. Such an approach is feasible due to seL4's small code size (compared with other operating system kernels), at around 8 700 lines of C code. Despite this fact, it is, to our knowledge, still the largest code base where a full context-aware WCET analysis has been performed.

Section 2 details the features of seL4 that make it amenable to automated analysis. Section 3 describes the methods used to analyse seL4. Section 4 shows the results of the analysis, outlining the worst-case execution paths found.

2 seL4 Design Features

The seL4 microkernel has several properties that assist with automated static analysis. First and foremost is that its code base is small. We analysed the ARM version of the seL4 kernel, which has around 8 700 lines of C code and 600 lines of ARM assembly code. Although this is a large body of code by WCET analysis standards, we found it to be just within the scalability limit for the implicit path enumeration technique (IPET) [LMW95]. The full analysis takes two hours to compute, and is described further in Section 3.

seL4 is an event-based kernel, where a single kernel stack is shared by all user threads. Context switching between user threads is performed by changing a variable containing the currently running thread. In contrast, process-based kernels, with dedicated per-thread kernel stacks, must switch the stack pointer during a context switch. This model may be more efficient in the presence of frequent context switches [Lie93b], but the event-based model of seL4 aids static analysis significantly, as control flow is more structured.

Code size	98704 bytes
Lines of code	8642
Number of functions	84
Number of basic blocks	1922
Number of loops	68
Number of branches	1410

Table 1: Properties of the analysed seL4 binary.

Other features that simplified our analysis are listed below. Many of these arose due to requirements of the formal verification process, without any regard to a WCET analysis.

- seL4 never stores function pointers at run-time, so all jumps can be resolved statically (with the help of symbolic execution).
- seL4 never passes pointers to stack variables. This simplifies the analysis of memory aliasing for WCET.
- The task of memory allocation is delegated to userspace, avoiding complex allocation routines within the kernel.
- There are very few nested loops within seL4 – automatically identifying nested loops at the assembly level and their loop relations is not an easy task in the presence of heavy compiler optimisations.
- Unbounded operations (such as object deletion) contain explicit preemption points. If an interrupt is pending at a preemption point, seL4 will postpone the current operation and return to a safe point to handle the interrupt.

seL4 is accompanied by a large body of machine-checked proofs which contains thousands of invariants and lemmas. It should be possible to incorporate these into a WCET analysis to assist in excluding many infeasible paths.

One issue that arose during the analysis of seL4 is that in two places mutually-recursive functions are used. The formal proof guarantees termination and actually proves that the functions do not call themselves more than once. This knowledge makes the analysis easier, as we could simply virtually inline each function at most twice. However, for this analysis, we chose to unwind the recursion manually.

The design of seL4, in conjunction with formally-proven guarantees, has greatly assisted in performing an automated static analysis.

3 Analysis Method

We performed a static analysis of the seL4 kernel binary to compute a safe upper bound of its WCET. For comparison, we constructed the worst-case scenarios detected by the analysis and executed them on real hardware. This gives a indication of how tight the analysis is. Table 1 summarises the relevant properties of the code analysed.

3.1 Static Analysis

We analysed seL4 for its interrupt latency by examining the worst-case execution time of all possible paths through the kernel, accounting for preemption points. Non-preemptible paths can begin at a number of places, such as entry to a system call or page-fault handler. Interrupts can be processed only once control is returned to the user. In seL4, explicit preemption points detect if an interrupt is pending within a long-running loop and if so, postpones the current operation and returns up the call stack. The interrupt latency is the sum of the WCET of the longest kernel path and the time taken to dispatch the interrupt to a user thread.

The seL4 binary we analysed was compiled with gcc using `-O2` optimisation level and additionally the `-fwhole-program` flag, which enables gcc to perform very aggressive optimisation and inlining of code. This means that most function boundaries are lost and functions are on average much larger because of inlining. The compiled binary also exhibits optimisations such as tailcalls and loop rotation.

Despite having well structured code, seL4 violates this structure in one specific code path. seL4 features a highly optimised routine for handling the most common IPC operations, known as the *IPC fastpath*; it improves the average time for these IPC operations by an order of magnitude. It does this using a continuation-based control flow, avoiding the need for stack unwinding. Unfortunately, the analysis tool currently does not support continuations—it expects all functions to return. As a result, we needed to disable the IPC fastpath at compile time. However, we do not expect this to affect our analysis, given the aforementioned presence of order-of-magnitude slower operations elsewhere in the microkernel.

The control flow graph (CFG) of seL4 is extracted from the binary, using symbolic execution to resolve indirect branches (via a register) and jump tables generated by switch statements. This step was performed without any user guidance, made possible by the absence of function pointers in seL4's sources.

The iteration counts of loops were specified by hand. Most have fixed bounds and could have been determined automatically with a rudimentary analysis. Some, however, depend on the state of the system—e.g. the number of runnable threads. These properties are all bounded by total physical memory. To support this we allow the user to provide an expression relating the iteration count to constants such as the size of physical memory. Due to heavy inlining by the compiler, none of the iteration counts in the binary are context-sensitive, even though some are at the source level (e.g. memcpy).

The control flow graph, along with the loop iteration counts, is passed to a modified version of Chronos 4.0, from NUS [LLMR07]. We adapted Chronos to support the ARM processor. Chronos uses the IPET method [LMW95], which converts the control flow graph into a system of linear equations (or inequalities) with integer coefficients. Chronos extends the basic IPET model with support for instruction caches and pipeline modelling. All function calls are virtually inlined so that the analysis is context-aware. This inlining results in almost 400 000 CFG nodes (basic blocks) in the analysis.

The output of Chronos is a system of linear constraints and an objective function to maximise subject to those constraints. With 400 000 CFG nodes, it creates two million variables and 2.5 million equations.

Finally, an off-the-shelf integer linear programming solver is used to compute the final WCET value. We used IBM's ILOG CPLEX Optimizer to compute the solution. This is the most computationally intensive step of the process, and takes up to two hours for the entire seL4 kernel, when performed on an Intel Core 2 Duo running at 2.93 GHz. However, smaller portions of the kernel are solved much faster—typically within a minute or less.

3.2 Hardware Measurements

Our test platform for measurements is a Beagleboard-xM with a TI DM3730 processor. This processor has an ARM Cortex-A8 core running at 800 MHz, with a 32 KB L1 instruction cache and a 32 KB L1 data cache, both 4-way set-associative. The experiments were configured to use 128 MB of physical memory. The latency of a read or write to physical memory on this platform was measured to be 80–100 cycles.

The L1 caches on the Cortex-A8 have an unspecified random replacement policy. This makes simulating the exact cache behaviour impossible, and effectively forces any safe cache analysis to assume a direct-mapped 8 KB cache. Furthermore, it makes it infeasible to construct a true worst-case scenario on hardware.

The Cortex-A8 has a dual-issue pipeline, which is not accounted for in our processor model. Whilst it is in theory possible to force the Cortex-A8 to single issue, this oddly requires a "high security" version of the processor which is not readily available. This means that we can expect the observed results to be up to 2x faster than computed by static analysis. Extending the static analysis model to support a dual-issue pipeline is the subject of future work.

The Cortex-A8 also supports speculative prefetching and branch prediction. These features were disabled in order to make measurements more deterministic.

Our experiments also disabled the data cache and L2 cache during both estimation and real execution, as our analysis tools do not yet support these on the ARM platform. This allowed us to confidently validate our timing model.

3.3 Open vs. Closed Systems

We analyse seL4 for two different use-cases—open systems and closed systems. We define an *open system* to be one where the system designer cannot prevent arbitrary code from executing on the system. This is in contrast to a *closed system*, where the system designer has full control over all code that executes.

In an open system, real-time subsystems may execute in conjunction with arbitrary and untrusted code (although confined by the capabilities provided to them). seL4 uses a strict priority-based round-robin scheduler. In such a scheme, time sensitive threads must be assigned the highest priority on the system so that they may run as soon as required (typically

System call	Description
`seL4_Send()`	Blocking send to an endpoint.
`seL4_Wait()`	Blocking receive on an endpoint.
`seL4_Call()`	Combined blocking send/receive.
`seL4_NBSend()`	Non-blocking send to an endpoint (fails if remote is not ready).
`seL4_Reply()`	Non-blocking send to most recent caller.
`seL4_ReplyWait()`	Combined reply and wait.
`seL4_Notify()`	Non-blocking send of a one-word message.
`seL4_Yield()`	Donate remaining timeslice to a thread of the same priority.

Table 2: System calls permitted in a closed system.

when triggered by a hardware interrupt). seL4's design disables interrupts whenever in the kernel, except at a few select preemption points. As a result, the interrupt latency for the highest-priority thread is determined by the worst-case execution time of all possible operations performed by seL4.

In a closed system, the system designer has full control over all operations performed by the kernel. Therefore she can ensure that operations that are known to be long-running do not occur at critical times, e.g. by allocating all resources at boot time and avoiding delete operations at run time. The interrupt latency in this scenario is defined by the WCET of a select number of paths within the kernel which are used by the running system—primarily inter-process communication (IPC) operations, as well as thread scheduling. The permitted system calls are listed in Table 2.

Note that `seL4_Call()` can be invoked on an IPC object to perform IPC operations, but invoking it on other object types may lead to the creation or deletion of kernel objects. We exclude these latter operations from the analysis of closed systems, allowing only the IPC-related uses of `seL4_Call()`.

4 Experimental Results

4.1 Open System

In an open system, the analysis pointed us to two interesting cases which were clear candidates for the worst-case execution path in seL4.

The first case arises due to the nature of IPC in seL4. Threads do not communicate with each other directly. Rather, they construct IPC "endpoints" which act as communication channels between threads. Multiple threads are permitted to wait to receive (or send) a message on a single endpoint—threads join a queue and are woken in turn as partners arrive. If the endpoint is deleted whilst there are still multiple threads waiting, each of these threads is removed from the endpoint queue and added to the scheduler's run queue. A malicious program (looking to force a deadline miss), could allocate as many threads as possible and construct this scenario. We constructed such a scenario with 91 000 threads (limited by physical memory). The results are shown in Table 3.

The second case arises due to a scheduler optimisation used in seL4 known as lazy scheduling [Lie93b]. In microkernel-based systems where IPC is frequent, a thread blocking on an IPC operation will often be made runnable again before the scheduler even needs to reconsider it for execution. To benefit from this observation, seL4 does not immediately remove threads from the run queue, but defers that work until a thread is selected to be scheduled. This leads to the obvious worst-case scenario where many non-runnable threads pollute the run queue. The scheduler must iterate over all of these threads, inspect and then dequeue them, until it finally finds a runnable thread (or the idle thread).

Case	Computed	Observed	Ratio
Endpoint deletion	215.3 ms	152.8 ms	1.41
IPC (open system)	566.2 ms	272.8 ms	2.08
IPC (closed system)	208.3 μs	118.2 μs	1.76

Table 3: Computed upper bound versus measured observations for feasible worst-case paths with data caches disabled.

We constructed this scenario, using the `seL4_TCB_Suspend()` operation which suspends a thread but does not immediately dequeue it from the run queue. The second row of Table 3 compares our computed value with measurements observed on hardware. In this case, a system with 128 MB memory can create 119720 threads.

4.2 Closed System

Within a closed system, where only the system calls outlined earlier in Table 2 are permitted, our analysis detects an infeasible worst-case scenario. The `seL4_Reply()` operation is used to respond to the most recent message received with `seL4_Wait()`. A one-time endpoint used to respond to the most recent sender (known as a *reply cap*) is stored in a dedicated location in each thread control block (the *reply slot*). The kernel must delete the existing reply cap before any call to `seL4_Wait()` and after a call to `seL4_Reply()`.

The analysis detected that deleting this reply cap could lead to a long delay at the next reschedule, for the same reasons as outlined in the first scenario of the open system, described earlier. Even though we excluded explicit delete operations from our analysis, this implicit operation was exposed. However, it is impossible to construct this scenario, as reply caps can only be used by other threads if they are first removed from the reply slot. Therefore the delete operation on the reply slot will only affect the schedulability of one thread.

With this knowledge, we could add an extra constraint which excluded this infeasible path. The new analysis determined that all IPC send or receive operations became candidates for the new worst-case path. It identified two factors which affect the IPC operation's execution time. The first is that endpoints are addressed using a structure resembling guarded page tables [Lie93a]; decoding the address involves traversing a graph up to 32 levels deep. The second is, unsurprisingly, the size of the message to be transferred, on which seL4 places a hard limit of 120 32-bit words. The combination bounds the worst-case interrupt latency of a closed system to a very reasonable limit.

This case was also reproduced on hardware using `seL4_ReplyWait()` to trigger it. The results are shown in the final row of Table 3.

4.3 Analysis of Results

Table 3 shows that there is a factor of up to 2.08x between the observed and computed execution times. This disparity can be attributed to both the random cache replacement policy of the instruction cache, as well as the dual-issue pipeline of the Cortex-A8. With a random cache replacement policy, constructing a true worst case on hardware is extremely difficult. Modelling the Cortex-A8 pipeline perfectly is also a difficult task. Given that the memory access latency on fast processors far outweighs the impact of pipeline effects, a simpler pessimistic pipeline model is sufficient. None of these factors cause the static analysis to be unsound, and therefore the computed values can be confidently used as a safe upper bound for hard real-time systems.

It should be noted that these results are much worse than reality as the data cache has been disabled both on hardware and in the model. As memory latency is up to 100 cycles on this platform, this adds a significant factor to the execution time of these test cases.

Certain code paths are guaranteed by the formal proof never to execute. These paths could potentially be pruned by incorporating invariants from this proof into the WCET analysis.

5 Conclusions and Future Work

As the trend of feature integration in embedded devices continues to gain momentum, integrating numerous complex software stacks in a fault-tolerant manner will be a necessity. In this paper, we assert that microkernels can be used as the basis for hard real-time systems that nonetheless feature such integration. The primary requirement placed on these microkernels is a reasonable guarantee on their interrupt latencies.

A tight static analysis of a microkernel to determine safe WCET bounds is in fact feasible, as demonstrated by our analysis of seL4. There are many features of seL4 that both ease the analysis process and reduce the interrupt latency, without the need for a fully preemptible kernel.

For the feasible paths in seL4, the disparity between our calculations and measurements arises for two reasons: first, the non-determinism of the target hardware, and second, surmountable limitations of our analysis tools.

Future work will focus on adding support for the data cache, and automatically incorporating proof invariants into the WCET analysis to further tighten the computed upper bound.

At present, seL4 can be used in a closed system with reasonably small guaranteed response times. In an open system, allowing untrusted code to execute, the response time guarantees are still bounded but too large to be useful, and highlight areas where seL4's real-time behaviour can be improved.

References

[CEE+02] Martin Carlsson, Jakob Engblom, Andreas Ermedahl, Jan Lindblad, and Björn Lisper. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *2nd International Workshop on Real-Time Tools*, 2002.

[CP01] Antoine Colin and Isabelle Puaut. Worst case execution time analysis of the RTEMS real-time operating system. In *13th ECRTS*, pages 191–198, Delft, Netherlands, Jun 13–15 2001.

[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 *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KWRP05] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using measurements as a complement to static worst-case execution time analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec 2005.

[Lie93a] Jochen Liedtke. A high resolution MMU for the realization of huge fine-grained address spaces and user level mapping. Arbeitspapiere der GMD No. 791, German National Research Center for Computer Science (GMD), Sankt Augustin, Germany, 1993.

[Lie93b] Jochen Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Asheville, NC, USA, Dec 1993.

[Lie96] Jochen Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. In *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, volume 69(1-3), December 2007.

[LMW95] Yau-Tsun Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[MHH02] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *23rd RTSS*, Austin, TX, USA, 2002.

[MHSH01] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. RTLinux with address spaces. In *3rd Real-Time Linux WS*, Milano, Italy, nov 2001.

[PZH07] Stefan M. Petters, Patryk Zadarnowski, and Gernot Heiser. Measurements or static analysis or both? In *7th WS Worst-Case Execution-Time Analysis*, Pisa, Italy, Jul 2007. Satellite WS 19th ECRTS.

[SEGL04] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, October 2004.

[SP07] Mohit Singal and Stefan M. Petters. Issues in analysing L4 for its WCET. In *1st MIKES*, Sydney, Australia, Jan 2007. NICTA.

[YB97] Victor Yodaiken and Michael Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997.