Timing Analysis of a Protected Operating System Kernel

Bernard Blackham[†], Yao Shi[†], Sudipta Chattopadhyay[‡], Abhik Roychoudhury[‡] and Gernot Heiser[†]

[†] *NICTA and University of New South Wales, Sydney, Australia*
[‡] *National University of Singapore*
Email: bernard.blackham@nicta.com.au

***Abstract*—Operating systems offering virtual memory and protected address spaces have been an elusive target of static worst-case execution time (WCET) analysis. This is due to a combination of size, unstructured code and tight coupling with hardware. As a result, hard real-time systems are usually developed without memory protection, perhaps utilizing a lightweight real-time executive to provide OS abstractions.**

This paper presents a WCET analysis of seL4, a third-generation microkernel. seL4 is the world's first formally-verified operating-system kernel, featuring machine-checked correctness proofs of its complete functionality. This makes seL4 an ideal platform for security-critical systems. Adding temporal guarantees makes seL4 also a compelling platform for safety- and timing-critical systems. It creates a foundation for integrating hard real-time systems with less critical time-sharing components on the same processor, supporting enhanced functionality while keeping hardware and development costs low.

We believe this is one of the largest code bases on which a fully context-aware WCET analysis has been performed. This analysis is made possible due to the minimalistic nature of modern microkernels, and properties of seL4's source code arising from the requirements of formal verification.

***Keywords*-Real time systems; Operating system kernels; Software verification and validation;**

I. Introduction

Hard real-time systems demand guaranteed interrupt response times, irrespective of system state when an interrupt arrives. Designers of complex embedded systems often separate out hard real-time components onto separate processors with deterministic hardware and minimal interference from other components. As manufacturers strive to gain a competitive advantage by adding features to embedded devices, these systems will need to integrate even more software components. The added complexity and cost of multiple dedicated processes does not scale—for example, cars and aircraft are trending towards combining both critical and convenience functionality; the cost, weight and power consumption of tens or hundreds of processors is a serious issue.

An alternative approach is to consolidate these systems onto a single processor, trusting a supervisor such as a microkernel or hypervisor to provide functional and temporal isolation between critical real-time components and less critical time-sharing components [1]. For hard real-time designs, this approach depends on the ability to provide safe upper bounds on the interrupt latency of the operating system (OS). Computing bounds for the interrupt latency of OS kernels is a task made difficult by their unstructured code, tight coupling with hardware and sheer size.

Using a microkernel-based design reduces the size and complexity of the analyzed code base. Static analysis can also be aided by structuring the kernel to avoid potential pitfalls that complicate the analysis. This paper analyzes the temporal properties of the seL4 microkernel, with a view to tuning the kernel for hard real-time applications.

seL4 [2] is a third-generation microkernel, broadly based on the concepts of L4 [3]. It provides virtual address spaces, threads, synchronous and asynchronous communication, and capabilities for managing authority. The distinguishing feature of seL4 is that it is the only kernel to date to be formally verified, with a formal machine-checked proof that the C code implementation adheres to the specification of the kernel. Among others, these proofs guarantee that seL4 will never crash or perform a (functionally) unsafe operation. Whilst these strong *functional* guarantees are sufficient for many systems, critical real-time systems also require *temporal* guarantees to achieve safety.

In this paper, we present a case-study of seL4 and analyze its worst-case execution time (WCET) and thus interrupt latency. seL4 has several properties that assist in the analysis and make it possible to give tighter bounds on execution time. Its structure helps to resolve many of the difficulties encountered in previous analyses of kernels.

It is possible to achieve very good interrupt latencies by making the kernel fully preemptible. In such a 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 approach makes it very difficult to reason formally (or even informally) about the behavior of the kernel, and adds complexity to development and testing. It requires very careful coding of the interrupt paths, and defensively analyzing that at every point in the kernel an interrupt cannot crash the kernel or make its state inconsistent.

Analyzing concurrency issues within a fully-preemptible kernel is extremely challenging due to the explosion of

possible interleavings to consider and the difficulty in reproducing timing-related bugs. Formal verification of a fully-preemptible kernel has not yet been achieved although there has been some progress [4].

In contrast, seL4 executes with interrupts disabled during kernel execution, and instead checks for pending interrupts at explicit preemption points. This greatly simplifies the design, testing and verification of the kernel, but increases its interrupt latency. Furthermore, locks are expensive—on modern hardware they can have significant cost even when not contended. As the most frequently-used seL4 primitives are fast (100s to 1000s of cycles), and preemption points can be used to limit uninterruptible execution of longer-running operations, this approach presents a tunable trade-off between worst-case and average-case performance. The latter is important for a general-purpose platform like seL4, and is particularly relevant for battery-powered hybrid (real-time plus best-effort) systems.

Furthermore, embedded processors are becoming faster, and latencies of $10,000$s to $100,000$s of cycles are acceptable in many situations. For example, 1 GHz processors are increasingly common on modern high-end embedded devices and can execute $100,000$ cycles in $100\,\mu$s—adequate for many applications. Hence, a fully preemptible kernel is generally not necessary to meet real-time requirements (except where ultra-low latencies are required) provided that the kernel can deliver reasonable interrupt latency guarantees.

We assert that a well-designed microkernel can be a suitable platform for supporting memory-protected hard real-time systems, even without being fully preemptible (and thus remaining suitable for performance- or energy-constrained hybrid applications). We analyze the timeliness of seL4, and its suitability for such use.

A. Contribution

This paper presents the first full interrupt-response-time analysis of an OS kernel providing full virtual memory and memory protection. We show that realistic safe upper bounds on interrupt latencies can be determined for protected microkernel-based systems. We perform a full *context-aware* WCET analysis of all of seL4's code paths on a modern embedded CPU architecture using a realistic pipeline model. By virtually inlining all functions in seL4, every possible calling context of each function is considered. Such an approach is feasible due to seL4's small code size (compared with other protected 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. We also measure the identified worst-case paths on real hardware, demonstrating that the WCET bounds we obtain are not overly pessimistic and can be used in practical systems.

Section II summarizes the state of the art of WCET studies of operating systems code. Section III details the properties of seL4 that make it amenable to automated analysis. Section IV describes the methods and tools used to analyze seL4. Section V gives some background on work performed to improve seL4's temporal behaviour. Section VI shows the results of the analysis, outlining the worst-case execution paths found.

II. Related work

There have been several studies on the interrupt latency of operating systems code, however none have successfully analyzed a kernel providing full virtual memory and memory protection.

Safe upper bounds for WCET are generally computed using a combination of static analysis techniques and measurements on real hardware [5], [6]. WCET bounds based on measurements alone cannot be relied upon—for example, measurement-based upper bounds stated for RTLinux [7] were later shown to be invalid [8].

The first major static WCET analysis of a real-time executive was published by Colin and Puaut [9], who analyzed the RTEMS operating system. They used a tree-based static analysis tool, HEPTANE, targeting a Pentium processor. They encountered issues such as unstructured loops with goto statements and function pointers, which all had to be resolved manually to complete the analysis.

A WCET analysis of the microkernel used by the OSE delta operating system was undertaken by Carlsson et. al. [10] and Sandell et. al. [11]. The OSE kernel permits interrupts in most kernel code, except within 612 "disable-interrupt" regions. Despite the complex control flow of the kernel, most of the disable-interrupt regions were simple, well structured and free of function pointers, simplifying the analysis.

The μC/OS-II kernel was analyzed for the WCET of each system call [12]. The analysis was generally successful, but required a significant amount of manual intervention.

The above systems are unprotected, single-mode kernels. An analysis was attempted on the L4 Pistachio kernel [13], which does offer memory protection. Unstructured code, inline assembly and context switching contributed to making the analysis difficult. Safe WCET bounds were never established. A more detailed summary of past RTOS analyses has been recently published by Lv et al. [14]

III. 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 analyzed the ARM version of the seL4 kernel, which has around $8,700$ lines of C code and 600 lines of ARM assembly code. The analysis was performed using a modified version of Chronos 4.1 [15], adapted for the ARM architecture, and took around four hours to compute. The analysis is described further in Section IV.

seL4 is an event-based kernel—i.e., the kernel uses a single kernel stack irrespective of which thread it is servicing. Context switching between threads is performed by changing a variable pointing to the currently running thread. In contrast, process-based kernels, with dedicated per-thread kernel stacks, switch the stack pointer during a context switch. As a result, the entire call stack is invalidated, and execution resumes based on the call stack from a prior context switch. The process-kernel model was originally thought to be more efficient in the presence of frequent context switching [16]. However, results on modern ARM hardware have shown the difference to be negligible and in fact event-based kernels have been shown to perform better in macro benchmarks [17]. The event-based model of seL4 aids static analysis, as control flow is more structured.

The seL4 microkernel was designed from its inception to be formally verified. Although the executable code is written in C, seL4 was initially developed in the functional language Haskell, to support rapid prototyping and provide a common ground for both the development team and verification team [18]. From the Haskell prototype, the development team implemented a high-performance C version, while the verification team mathematically formalized the expected behavior of the kernel into an *abstract specification*. The correctness of seL4 relies on a chain of proofs that the C code implements the Haskell prototype, and that the Haskell prototype conforms to the abstract specification.

As a result of this construction, the requirements of formal verification, as well as the security-driven desire for strong resource isolation, the C implementation of seL4 exhibits a number of properties that simplified our analysis:

- seL4 never stores function pointers at run-time. This allows all branches to be automatically resolved offline (with the help of symbolic execution).
- seL4 never passes pointers to stack variables. This eliminates the possibility of variables aliasing stack memory, simplifying the analysis of memory aliasing for WCET.
- The task of memory allocation is delegated to userspace, avoiding complex allocation routines within the kernel. The kernel checks that regions do not overlap but these checks are much simpler than the code for a complete allocator.
- 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, or even well-defined, in the presence of heavy compiler optimizations.
- 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 up the call stack to a safe context to handle the interrupt.

Table I
PROPERTIES OF THE ANALYZED SEL4 BINARY.

Code size (bytes)	42,120
Number of instructions	10,271
Number of functions	228
Number of basic blocks	2,384
Number of loops	56

It is worth noting that many of these properties arose because of requirements of the formal verification process, without any regard to a WCET analysis. Despite the restrictions imposed by some of these properties, seL4 does not suffer a significant performance penalty—its hot-cache IPC performance is within 10% of a heavily optimized assembly IPC path in OKL4 2.1 [2].

seL4's specification dictates that the kernel will never enter an infinite loop—i.e., all seL4 system calls eventually return to the user. This allows the analysis to ignore any infinite loops that exist in the kernel. Such loops are executed when assertions in the C code fail, however the proofs guarantee that these assertions are always true (under the assumptions of the proof, which include a correct C compiler and the absence of hardware bugs).

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

In seL4's event-based model, almost all functions return to their caller, making static analysis simpler. However, this is not true in one specific code path: seL4 features a highly optimized C 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 so by using a continuation-based control flow, avoiding the need for stack unwinding. The analysis toolchain required some work to support a continuation-based control flow as it previously assumed that all functions would return.

Finally, 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. They have not been utilized in this analysis, but are the subject of future work.

Some interesting statistics of the analyzed seL4 binary are summarized in Table I.

IV. ANALYSIS METHOD

We performed an analysis of the seL4 kernel binary to compute a safe upper bound on its interrupt latency. For

comparison, we constructed scenarios to exercise the worst-case paths detected by the analysis and executed them on real hardware. This gave us an indication of how closely the computed bounds reflected reality.

A. Processor Model

seL4 can run on a variety of ARM architectures and platforms. For this analysis we chose the BeagleBoard-xM platform with a TI DM3730 processor. This processor has an ARM Cortex-A8 core running at 800 MHz, with separate 32 KiB L1 instruction and data caches, both 4-way set-associative. The L2 cache was not modeled in this analysis as our tools do not yet support this. As a result, the L2 cache was also disabled in hardware for our measurements.

The experiments were configured to use 128 MiB of physical memory. The latency of a read or write to physical memory on this platform was measured to be 80–100 cycles; in the static analysis we assume a 100-cycle penalty for all cache misses. The seL4 kernel locks its pages into the TLB so that there are no TLB misses during execution.

The Cortex-A8 has a 13-stage dual-issue pipeline, with in-order issue, execution and retirement. Most arithmetic instructions[1] can be issued simultaneously with a subsequent arithmetic instruction or memory load, provided that there are no dependencies between them. Forwarding paths between stages permit single-cycle instructions to execute without stalls arising from register dependencies. The compiled seL4 binary happens to use only those arithmetic instructions which can be dual-issued—in particular, there are no multiplication or "`rrx`" operations which would incur a multi-cycle latency. Our static analysis models the dual-issue nature of the pipeline for arithmetic instructions.

All non-arithmetic instructions executed by the seL4 kernel fall into one of the following categories:

- load/stores—these incur a 100 cycle memory access penalty on a cache miss or with caches disabled;
- branches—these incur a 13 cycle penalty without prediction (see below);
- coprocessor operations (`mcr`/`mrc`)—these include cache flushing, TLB flushing and address space switching which can take up to 120 cycles and depends on the particular operation and the state of the system;
- synchronization instructions (`isb`/`dsb`)—these take up to 13 or 29 cycles, respectively;
- processor state changes (`cps`)—these take 60 cycles.

The Cortex-A8 also supports speculative prefetching and branch prediction. These features were disabled in hardware in order to make measurements more deterministic, and were not included as part of the processor model.

The L1 caches on the Cortex-A8 have an unspecified random replacement policy. This prevents simulation of the exact cache behavior, and effectively forces any safe cache analysis to assume a direct-mapped 8 KiB cache. Furthermore, it makes it infeasible to construct a true worst-case scenario on hardware, Thus we can only determine an upper bound on the pessimism of our model.

[1]On ARM the "arithmetic instructions" also include logical primitives. The full list is: `ADC`, `ADD`, `AND`, `BIC`, `EOR`, `ORR`, `RSB`, `RSC`, `SBC`, `SUB`, `CMN`, `CMP`, `TEQ`, `TST`, `MOV` and `MVN`.

B. Static Analysis

We analyzed seL4 for its interrupt latency by examining the worst-case execution time of all possible paths through the kernel. A path through the kernel begins at one of the kernel's exception handlers, such as the system-call or page-fault handler. A path ends when control is passed back to the user, or at the start of the kernel's interrupt handler code.

seL4 disables interrupts at all times when executing kernel code, but checks for pending interrupts at explicit preemption points (typically in long-running unbounded loops). If an interrupt is detected at a preemption point, seL4 postpones the current operation and immediately handles the interrupt. As preemption points occur only at the end of loops, we account for them in the analysis by forcing the iteration count of the loop to 1. The worst-case scenario for a path including a preemptible loop occurs when an interrupt arrives immediately after entry to the kernel. The kernel will still execute the operation with one iteration of the loop to ensure progress is made (avoiding livelock) and then handle the interrupt.

Interrupts arriving during kernel execution are handled immediately prior to returning to the user. The interrupt latency is therefore the sum of the WCET of any path through the kernel up to this point, and the time taken to dispatch the interrupt to a user thread.

The seL4 binary we analyzed is compiled with gcc 4.5 (from CodeSourcery's 2010.09 release) using the `-O2` optimization level and additionally the `-fwhole-program` flag, which enables gcc to perform very aggressive optimization and inlining of code. This means that most function boundaries are lost and functions are on average much larger because of inlining. Chronos has a feature to assist in correlating source code to instructions with the help of debug information. However, the structure of the output from our version of gcc bears very little resemblance to that of the source code, making this correlation difficult to automate. It may be possible to use the intermediate representations within gcc to automate the process, but we have not investigated this.

Figure 1 outlines the tools and workflow used to analyze seL4. We wrote a program to extract the control flow graph (CFG) of seL4 from the kernel binary. This step can be performed without any user guidance thanks to the absence of function pointers in seL4's C sources.

We use symbolic execution to extract the CFG, by finding all reachable instructions within the kernel from the given set of entry points (system call, page fault, invalid instruction, etc). Determining the destination of some branches requires

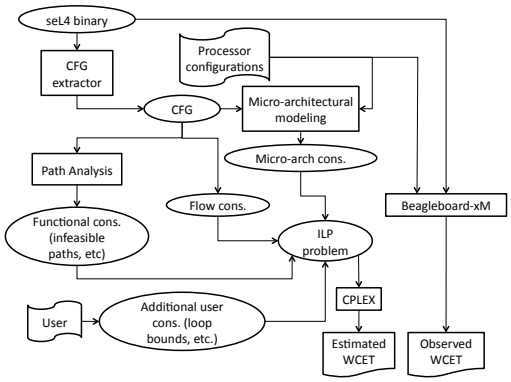

Figure 1. Workflow used to analyze seL4.

evaluating sequences of pointer arithmetic on read-write memory (e.g. return addresses pushed onto the stack). By symbolically executing the binary, we can resolve all of these off-line. The symbolic execution is not complete, but is sufficient to simulate any arithmetic operations, memory loads and stores which ultimately affect the possible destinations of a branch.

Loop bounds: We specified the iteration counts of the 56 loops by hand. Due to compiler optimizations, some effort was required to correlate loops in the binary CFG to the source code. Most loops in seL4 have obvious fixed bounds which can be determined automatically with a rudimentary analysis utilizing constant propagation. Some loops have iteration counts which would require more sophisticated analysis and reasoning—however there is sufficient information in the binary to ascertain all of these automatically.

All iteration counts within seL4 are fixed and independent of any system state. 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 analysis itself virtually inlines all function calls, so even if the compiler had not performed such aggressive inlining, context-dependent iteration counts could still be specified, or possibly computed automatically.

The control flow graph, along with the loop iteration counts, is passed to a modified version of Chronos 4.1 [15].

The compiled binary exhibited optimizations such as tail-calls (where a function returning up the call stack shortcuts intermediate callers when possible) and loop rotation (where the loop body is "rotated" so that the entry and exit points of the loop are not necessarily the first and last instructions of the loop body). This required some modifications to Chronos, as these optimizations violated its assumptions about the structure of functions and loops.

C. Static timing analysis by Chronos

We chose to use the Chronos tool for our analysis because of its existing support for instruction and data caches, its flexible approach to modeling processor pipelines, and its open license.

Static timing analysis through Chronos is broadly composed of two parts: i) micro-architectural modeling and ii) path analysis. Micro-architectural modeling involves estimating the timing effects of major micro-architectural components of the underlying hardware platform such as the pipeline and caches. Path analysis exploits infeasible path information in the control flow graph to estimate the execution time of the longest program path.

We adapted Chronos to support the ARMv7 instruction set. Precise modeling of the Cortex-A8 pipeline is made difficult not only by its complexity, but also due to a lack of sufficient documentation. Even the official documents from the manufacturer have been shown to be inaccurate [19]. We therefore used a conservative approximation of the pipeline, as described in Section IV-A.

Chronos models the pipeline on the granularity of a basic block. For each basic block, it constructs an execution graph. Nodes in the execution graph correspond to the different stages of pipeline for each instruction in the basic block. Edges in the execution graph correspond to dependencies among pipeline nodes. Such a flexible modeling of pipeline behavior via dependence edges also allows Chronos to model advanced processor features such as superscalarity.

For a normal processor, we have dependence edges between the pipeline stages of subsequent instructions, for example $IF(I_{i+1})$ can only start after $IF(I_i)$ concludes, where I_{i+1} is the instruction following I_i and IF denotes instruction fetch. For a superscalar processor with n-way fetch we set dependency edges between $IF(I_{i+n})$ and $IF(I_i)$, thereby allowing greater parallelism. The estimation method supports any execution graph; various pipeline models can be achieved by simply altering the automated mechanism for generating the execution graph from basic blocks.

Instruction cache modeling can be easily integrated with the pipeline modeling in Chronos. Chronos categorizes each instruction memory block m into one of the following: if m is categorized as *HIT*, then the memory block is always a cache hit. If m is categorized as *MISS*, then the memory block is always a cache miss. Finally, if m is categorized as *UNCLEAR*, static cache analysis has failed to determine an accurate cache categorization of m.

The instruction cache is accessed at the IF stage. Pipeline modeling computes an interval of each pipeline stage for each instruction in the basic block. If an instruction is *HIT*, then we can add [1,1] as the instruction cache latency to the computed interval of the IF node. If an instruction is a *MISS* and miss penalty is *lat*, we add [*lat*, *lat*] to the computed interval. Finally, if the instruction cache categorization is *UNCLEAR*, the exact latency is not known but we are sure

that it is between the interval [1,*lat*]. Therefore, we add [1,*lat*] to the computed interval.

A major source of imprecision in data-cache analysis comes from the "address analysis"—estimating the set of memory addresses touched by an instruction. Data-cache analysis in Chronos avoids this by determining which instances of a given instruction touch the same memory address. This leads to an analysis framework which is scalable, precise and also takes into account program scopes.

Chronos differentiates the cache contexts in terms of loop nests. For a single loop nest, Chronos considers two different contexts: the first iteration and all subsequents iterations. This contextual cache analysis greatly helps to give tighter estimates as the cache categorization of a memory block may vary significantly, depending on its loop nest.

After the micro-architectural modeling, we get the WCETs of each basic block, which are fed to the program path analysis stage. Program path analysis in Chronos is performed via integer linear programming (ILP) based on the implicit path enumeration technique (IPET) [20]. The basic approach is to view WCET estimation as an optimization problem. We formulate a linear programming problem where the variables are the execution counts of the basic blocks, and the coefficients are the execution times of each basic block (obtained from micro-architectural modeling). The flow constraints from the control flow graph, the loop bounds, as well as any available/detected information about infeasible control flow paths are all encoded as linear constraints. The execution time taken by the program is defined as a linear function which is maximized subject to these linear constraints. We refer the reader to other papers for details of the static analysis employed by Chronos [21], and its techniques for data-cache modeling [22].

All function calls are virtually inlined in Chronos so that the cache analysis is context-aware. This results in approximately $400,000$ basic blocks after inlining. The output from Chronos is a system of linear constraints and an objective function to maximize subject to those constraints. With $400,000$ basic blocks, Chronos creates 2.3 million variables and three million equations.

Finally, we utilize an off-the-shelf ILP solver—IBM's ILOG CPLEX Optimizer 12.2—to solve the ILP problem generated by Chronos. The result gives the WCET value and the assignment of edge counts, which can be used to reconstruct a corresponding path. This is the most computationally intensive step of the process, and takes about four hours for the entire seL4 kernel, when performed on an Intel Xeon running at 2.66 GHz. Smaller portions of the kernel are solved much faster—within seconds for most. Section VI-D details the time taken to analyze portions of the kernel.

D. Hardware Measurements

In order to get an idea of the degree of pessimism in our WCET estimates, we measured actual execution times on hardware for the feasible worst-case paths detected by static analysis. For timings we used the Cortex-A8's cycle counter, which is a part of its performance monitoring unit.

The results from the ILP solver specify the execution counts of basic blocks and edges of the worst-case path. To determine the path itself, we construct a directed multigraph (a graph with potentially multiple edges between nodes) using the nodes of the original CFG, and replicate the CFG edges as many times as the execution count variable corresponding to that edge dictates. The path taken is then given by an Eulerian path from the source to sink—a path that traverses every edge once and only once. Such a path is guaranteed to exist in the graph because of the flow constraints for each node. Note that this path may not be feasible due to semantic behavior of the code which is not modeled in the static analysis.

In the absence of nested loops, this information is sufficient to uniquely identify the concrete path through the kernel. With nested loops, the edge counts in the ILP solution is not sufficient to uniquely identify a path, as inner loop iterations could potentially differ between iterations of the outer loop. In seL4, nested loops are sufficiently rare and consistent in structure—all inner loops where this arises have a fixed iteration count.

We manually constructed test cases to approximate the worst-case path resulting from static analysis. In some cases, these paths turned out to be clearly infeasible, in which case we manually added constraints to exclude these paths.

E. Comparing Static Analysis with Measurements

In order to compare our static analysis model with empirical results, we analyzed a number of test programs aimed to exercise various parts of the seL4 kernel API. The test cases were run on a simulator, QEMU, which provided us with a trace of the instructions that would be executed. We also ran the test case on hardware to obtain empirical timings. All test cases are deterministic and so always produce the same execution paths in the simulator as on hardware (assuming there are no bugs in the simulator).

Using the instruction trace, we determined the precise iteration counts of the loops as executed and provided these as extra linear constraints to Chronos. Finally, we verified that the new path from our static analysis matched what was executed on the simulator. Note that this was only used for determining the amount of pessimism in our model, and not for the safe WCET bounds themselves.

Figure 2 shows the difference between the estimated execution time and the real execution time for typical uses of the system calls in Table II. The average ratio of all system calls in Figure 2 is 5.98, and the largest ratio is 7.38. For each system call, because the estimated path and the executed path are identical, we attribute the error to conservatism in the pipeline and cache models and the

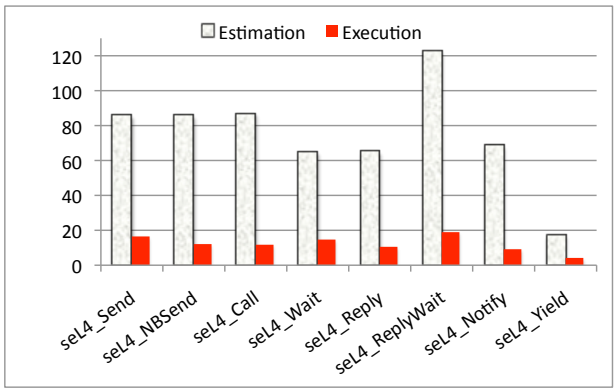

Figure 2. The error between estimation and real execution time for typical invocations of the seL4 system calls, measured in μs.

inability to exercise the worst-case cache/pipeline behaviour of the processor.

F. Open vs. Closed Systems

We analyzed 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. This is a coarse parameterization of the kernel, as suggested by Schneider [23], in order to achieve tighter bounds on the WCET in real-world use cases.

In seL4, all long-running system calls have preemption points to prevent unbounded execution with interrupts disabled. However, there still exist paths with a longer execution time than desirable for some hard real-time systems, and where adding preemption points is increasingly difficult to both implement and reason about formally. For example, adding preemption points to certain object creation and deletion paths would lead to the existence of partially constructed (or deconstructed) objects. These objects must be handled specially by other kernel operations and considered throughout the formal proof. This is the subject of future work on seL4, and leads us to differentiate between open and closed systems.

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 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 the sum of the worst-case execution time of all possible non-preemptible operations performed by seL4, and the time taken to handle and dispatch an interrupt.

Table II
SYSTEM CALLS PERMITTED IN A CLOSED SYSTEM.

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.

In a closed system, the system designer has full control over all operations performed by the kernel. Therefore she can ensure that operations which have a significant impact on interrupt latency are avoided at critical times (e.g. by allocating all resources at boot 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. Table II lists the permitted system calls.

Note that `seL4_Call()` can be invoked on an IPC object to perform IPC operations, but it is also used to interact with other kernel objects. We exclude these latter operations from the analysis of closed systems, allowing only the IPC-related uses of `seL4_Call()`.

V. INITIAL WCET RESULTS

Our initial analysis of seL4 pointed us to some serious latency issues under pathological circumstances. One such issue arose due to a scheduler optimization used in seL4 known as lazy scheduling [16]. 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 would not immediately remove threads from the run queue, but deferred that work until a thread was selected to be scheduled. Whilst this improved the common case, it led to a 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).

In conjunction with the seL4 development team, we removed lazy scheduling from the kernel. We replaced it by a model where only runnable threads reside on the run queue, *except* the actively running thread. This model avoids run queue manipulation for the most common IPC operations: if a running thread A performs a blocking IPC that wakes up a blocked thread B, then thread A moves from the running state to the blocked state, and thread B moves from the

blocked state to the running state. These transitions avoid the run queue altogether, thereby maintaining seL4's fast IPC performance.

We discovered other unbounded non-preemptible paths in our initial analysis. The primary culprits were in the object creation and deletion paths. We modified these to introduce additional preemption points so that their non-preemptible regions are bounded. The formal verification of these modifications has not yet been completed, however there are no insurmountable obstacles to doing so.

Combined with the removal of lazy scheduling, these modifications have reduced seL4's worst-case interrupt latency to a constant, independent of variables such as the number of objects on the system. The results presented in the following section use this updated kernel.

VI. Experimental Results

A. Open System

In analysing an open system, we consider all possible seL4 operations. Without any user input other than the loop bounds themselves, our toolchain determined the worst-case execution time to be in excess of 20 ms, and corresponded to an infeasible execution path. We excluded several infeasible paths by adding additional constraints on the set of ILP equations until we found a feasible worst case.

We found the single longest path to occur when deleting an *ASID pool*. ASID pools are seL4 objects used to manage collections of virtual address spaces. Deleting an ASID pool requires iterating over each address space in the pool (of which there may be 1024), and invalidating the user's TLB mappings associated with each one.

Beyond this case, we also found that most of the long non-preemptible paths occured when creating, deleting or recycling[2] objects. The largest non-preemptible loop in seL4, by measure of iteration count, is in one of these paths and iterates over a 16 KiB page directory in order to clear all of its associated mappings.

The other entry points into the kernel (unknown system calls, undefined instructions, page faults, interrupts), do not invoke the creation or deletion paths. Their worst cases involve a simple asynchronous message delivery to a specific thread on the system and are therefore very lightweight.

The results of these are shown in Table III.

B. Closed System

Within a closed system, we permit only the system calls outlined in Table II, along with page faults, undefined system calls and invalid instructions (commonly used by virtualization), and of course, interrupts. The only difference between the open and closed analysis is in the system-call handler. All other exception paths are identical to the open-system case.

[2]An seL4 *recycle* operation is equivalent to deleting and re-creating the object, but is faster and requires less authority than re-creating the object.

The most significant worst-case scenario that arose was the `seL4_ReplyWait()` system call. `seL4_ReplyWait()` is used to respond to the most recently received message, and then wait for a new incoming message. The particular scenario detected was infeasible and is described below. It is an interesting infeasible case as there are in fact invariants in the formal proof that could potentially be utilized to exclude this automatically.

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 an endpoint—threads join a queue and are woken in turn as partners arrive. Deleting an endpoint in such a state leads to a long-running operation as threads are unblocked and moved back onto the scheduler's run queue. Although this is done preemptibly, it still adds a non-trivial amount of work to this path. On closed systems, we do not permit a delete operation, so this scenario should not be considered.

However, `seL4_Reply()` and `seL4_Wait()` utilize a one-time endpoint (known as a *reply cap*) which is stored in a dedicated location in each thread control block (the *reply slot*). The kernel must delete the existing reply cap before a call to `seL4_Wait()` and after a call to `seL4_Reply()`.

The analysis detected that deleting this reply cap could lead to a longer execution time as it entered the deletion path. Even though we excluded explicit delete operations from our analysis, this implicit operation was exposed. However, it is impossible to construct this scenario, as a reply cap can only be used by other threads if it is first removed from the reply slot. Therefore the delete operation on the reply slot never needs to enter the longer deletion path.

With this knowledge, we added an extra constraint which excluded this infeasible path. The new analysis determined that the worst-case path for each kernel entry point is bounded by the time taken to perform an IPC.

In a feasible IPC operation, we identified three factors which affected execution time. The first is that endpoints are addressed using a structure resembling guarded page tables [24]; decoding the address involves traversing up to 32 edges of a directed graph. The second is, unsurprisingly, the size of the message to be transferred, on which seL4 places a hard limit of 120 32-bit words. Finally, an IPC may also grant the recipient access to seL4 objects, which requires additional bookkeeping to be done by the kernel. With these three factors exercised to their limits, the worst-case latency is still bounded to a reasonable value.

We reproduced this case on hardware and measured its execution time, as shown in Table III.

C. Analysis of Results

The results in Table III show that there is a factor of up to 10.15 between the observed and computed execution times. This disparity can be attributed to both the random

Table III
COMPUTED WCET VERSUS OBSERVED WCET FOR FEASIBLE WORST-CASE PATHS IN SEL4.

Event handler	Computed	Observed	Ratio
Syscall (open)	1634.8 μs	305.2 μs	5.36
Syscall (closed)	387.4 μs	46.4 μs	8.21
Unknown syscall	173.3 μs	17.9 μs	9.68
Undefined instruction	173.4 μs	17.1 μs	10.15
Page fault	175.5 μs	18.9 μs	9.27
Interrupt	104.7 μs	13.1 μs	8.01

Table IV
MOST COMPUTATIONALLY INTENSIVE SEL4 FUNCTIONS WHEN SOLVING FOR WCET.

	Function	BBs before inlining	BBs after inlining
1	arm_swi_syscall	2,384	433,085
2	handleSyscall	2,381	433,082
3	decodeInvocation	2,148	139,066
4	handleInvocation	2,237	140,890
5	decodeCNodeInvocation	985	32,106
6	decodeTCBInvocation	958	100,810

		Computation Time		
	Function	Chronos	CPLEX	Total
1	arm_swi_syscall	6m52s	227m37s	234m29s
2	handleSyscall	6m42s	226m9s	232m51s
3	decodeInvocation	2m49s	44m11s	47m0s
4	handleInvocation	2m20s	14m50s	17m10s
5	decodeCNodeInvocation	37s	4m59s	5m36s
6	decodeTCBInvocation	2m11s	2m14s	4m25s

cache replacement policy of the instruction and data caches, and conservatism in modeling the Cortex-A8 pipeline. It is extremely difficult to model worst-case scenarios for these without very fine control over the processor's state.

A proportion of the disparity can also be attributed to infeasible paths detected by the analysis. Although we manually excluded some infeasible paths, the final paths detected may still contain smaller portions that are not possible. There is scope here to improve the bounds further, to the degree permitted by the inherent unpredictability of the hardware.

To compute the worst-case interrupt latency of the system, from interrupt arrival to a userspace interrupt handler executing, we take the largest WCET value from the given scenario and add the WCET of delivering an interrupt to the highest priority thread on the system.

Overall, these results show that seL4 could be used as a platform for closed systems and provide a guaranteed interrupt response time of $387.4 + 104.7 = 492.1\,\mu$s on this hardware. In open systems, the interrupt response time is $1635\,\mu\text{s} + 104.7\,\mu\text{s} = 1.74\,\text{ms}$, which is still quite reasonable for many applications. However there is clearly scope to improve the response times for both open and closed systems.

D. Computation Time

To examine the scalability of our analysis method on seL4, we computed the worst-case execution time for not only the top-level entry points, but also all subroutines in the seL4 binary. Table IV shows the functions in seL4 that took the longest time to analyze. `arm_swi_syscall` is the assembly-level entry point for all system calls. It directly or indirectly calls the other functions listed in the table. All other functions within the kernel are solved much faster (in minutes or seconds, rather than hours). The table also shows the number of basic blocks (BBs) before and after virtual inlining, as a measure of how complex the computation is.

Only five functions took more than five minutes to solve. They contain the largest number of basic blocks of any function in the kernel, once virtual inlining has been performed. From this table, it can be seen that the approach used almost scales up to the size of seL4, depending on one's patience.

VII. CONCLUSIONS AND FUTURE WORK

We have presented an interrupt-response-time analysis of the seL4 microkernel, highlighting the properties of the code base that make it amenable to static analysis. This is the first analysis to our knowledge of a protected operating system providing virtual memory. There are many properties of seL4 that both ease the analysis process and reduce the interrupt latency, without the need for a fully-preemptible kernel. Our analysis shows that seL4's WCET can be kept low enough for many closed- as well as many open-system applications.

seL4 is designed as a highly-secure general-purpose OS kernel, suitable for a large class of real-time, best-effort and hybrid systems. One of the design decisions supporting these goals is to disallow interrupts in the kernel, and instead limit interrupt latencies via specific preemption points. This means that the kernel cannot support extremely low interrupt latencies (of the order of 10s or 100s of cycles). seL4's latencies are currently several hundreds of thousands of cycles in the worst case. With the introduction of additional preemption points, guided by this analysis, we believe seL4's interrupt latency can reduced to just thousands of cycles. Future work will focus on achieving this whilst maintaining its proofs of correctness.

There are still significant differences between the results of static analysis and measurements on hardware. A part of this is due to conservatism in the pipeline and cache model used by static analysis (mostly inevitable due to the degree of undocumented behaviour in the processor). Another factor is code paths which are difficult to reproduce from userspace because of very fragile pathological cases. It may be possible to set up the state of the relevant data structures from within the kernel to force these paths, but this may not result in a realistic scenario.

Finally, some amount of pessimism comes from undetected infeasible paths in the control flow graph. While we could eliminate some of these in our analysis, we expect that we can automate and improve this by making use of invariants proved during the formal verification of the kernel. There is also scope to automate the determination of

loop bounds so that user input is not required, removing a potential source of misinformation.

Despite these limitations, we have been able to perform a complete WCET analysis of seL4. In conjunction with a formal proof of functional correctness, this already makes seL4 a compelling platform for safety-critical applications.

Whilst the focus of this paper has been WCET analysis for determining interrupt response time of the kernel, the results of the analysis can also be used to determine bounds on the latency of individual kernel operations. Bounds for non-preemptible kernel operations are directly obtainable from the analysis presented, whilst bounds for preemptible operations need to consider the running system and its possible preemptions. This knowledge forms part of a schedulability analysis which is crucial for designing reliable hard real-time systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and feedback.

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.

REFERENCES

[1] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *23rd RTSS*, Austin, TX, USA, 2002.

[2] 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*, Big Sky, MT, USA, Oct 2009, pp. 207–220.

[3] J. Liedtke, "On μ-kernel construction," in *15th SOSP*, Copper Mountain, CO, USA, Dec 1995, pp. 237–250.

[4] X. Feng, Z. Shao, Y. Dong, and Y. Guo, "Certifying low-level programs with hardware interrupts and preemptive threads," in *PLDI*, Tucson, AZ, USA, Jun 2008, pp. 170–182.

[5] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*. UBooks Verlag, Dec 2005, vol. 2.

[6] S. M. Petters, P. Zadarnowski, and G. Heiser, "Measurements or static analysis or both?" in *7th WS Worst-Case Execution-Time Analysis*, Pisa, Italy, Jul 2007, satellite WS 19th ECRTS.

[7] V. Yodaiken and M. Barabanov, "A real-time Linux," in *USELINUX*, Anaheim, CA, January 1997, satellite WS 1997 USENIX.

[8] F. Mehnert, M. Hohmuth, S. Schönberg, and H. Härtig, "RTLinux with address spaces," in *3rd Real-Time Linux WS*, Milano, Italy, Nov 2001.

[9] A. Colin and I. Puaut, "Worst case execution time analysis of the RTEMS real-time operating system," in *13th ECRTS*, Delft, Netherlands, Jun 13–15 2001, pp. 191–198.

[10] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper, "Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system," in *2nd Int. WS Real-Time Tools*, 2002.

[11] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper, "Static timing analysis of real-time operating system code," in *1st Int. Symp. Leveraging Applic. Formal Methods*, October 2004.

[12] M. Lv, N. Guan, Y. Zhang, R. Chen, Q. Deng, G. Yu, and W. Yi, "WCET analysis of the μC/OS-II real-time kernel," in *12th Int. Conf. Computational Sci. & Engin.*, Vancouver, Canada, Aug 2009, pp. 270–276.

[13] M. Singal and S. M. Petters, "Issues in analysing L4 for its WCET," in *1st MIKES*. Sydney, Australia: NICTA, Jan 2007.

[14] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A survey of WCET analysis of real-time operating systems," in *ICESS*, Hangzhou, China, May 2009, pp. 65–72.

[15] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," in *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, vol. 69(1-3), Dec 2007.

[16] J. Liedtke, "Improving IPC by kernel design," in *14th SOSP*, Asheville, NC, USA, Dec 1993, pp. 175–188.

[17] M. Warton, "Single kernel stack L4," BE Thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Nov 2005.

[18] G. Klein, P. Derrin, and K. Elphinstone, "Experience report: seL4 — formally verifying a high-performance microkernel," in *14th ICFP*, Edinburgh, UK, Aug 2009, pp. 91–96.

[19] B. Avison, "ARM Cortex-A8 instruction timings," http://www.avison.me.uk/ben/programming/cortex-a8.html, visited 8 May 2011.

[20] Y.-T. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *16th RTSS*, 1995, pp. 298–307.

[21] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for WCET analysis," *Real-Time Syst.*, vol. 34, pp. 195–227, 2006.

[22] B. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for WCET estimation," in *17th RTAS*, Apr 2011.

[23] J. Schneider, "Why you can't analyze RTOSs without considering applications and vice versa," in *2nd WS Worst-Case Execution-Time Analysis*, 2002.

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