To Preempt or Not To Preempt, That Is the Question

Bernard Blackham, Vernon Tang and Gernot Heiser
NICTA and University of New South Wales, Sydney, Australia
{Bernard.Blackham,Vernon.Tang,gernot}@nicta.com.au

Abstract

Real-time operating systems (RTOSes) are traditionally designed to be fully preemptible. This improves the average interrupt response time of the system but increases kernel complexity. An alternative design is to make the kernel mostly non-preemptible and only handle pending interrupts at specific preemption points within the kernel. While this potentially worsens interrupt response times, we claim that for a *protected-mode* RTOS, as required for multi-criticality systems, non-preemptible kernels can achieve worst-case latencies comparable to those of fully-preemptible kernels.

In order to understand the latency limits achievable in both approaches, we analyse and compare the worst-case interrupt latencies of a fully-preemptible commercial RTOS (QNX Neutrino) and a non-preemptible real-time kernel (seL4). Our results indicate that a non-preemptible kernel can achieve interrupt latencies which are within a factor of two from those exhibited by a fully-preemptible kernel.

1 Introduction

Hard real-time systems demand predictable *worst-case* interrupt latencies—their interrupt response times must be both bounded and short enough for the application domain. This is especially true for the real-time operating system (RTOS) that supports such a real-time system. RTOSes, which traditionally do not employ memory protection or dual-mode execution, have met these requirements by a *fully-preemptible* design, where (almost) all RTOS code runs with interrupts enabled.

While minimising interrupt latencies, the preemptible design has a cost: RTOS code must be reentrant and is full of critical sections, which must be protected by locks. Not only do locks introduce run-time overhead (which affects average-case performance), the resultant highly-concurrent code is difficult to reason about and therefore highly error-prone, significantly increasing the potential for obscure bugs which can be extremely hard to find and eliminate.

The growing complexity of real-time systems, in particular the advent of *multi-criticality* systems, where multiple functions are provided on a single processor, necessitates the use of isolation between components (and the RTOS). In a protected-mode OS, hardware-imposed overheads significantly increase the cost of switching between application and OS operation, and the cost of switching between user contexts (e.g. to handle an interrupt destined for a partition different from the one presently executing).

In fact, these hardware costs are in the order of dozens or even hundreds of cycles, in the same magnitude as the (generally simple) functions performed by an RTOS. This means that the cost of preempting the RTOS operation is similar to the time required for letting it complete uninterrupted, resulting in a significant shift in performance trade-offs compared to classical, unprotected RTOSes.

Yet, the belief persists in academia and industry that an RTOS must be fully preemptible: for example, commercial RTOSes supporting memory protection, such as QNX's Neutrino [QNX], Wind River's VxWorks [Win] and Green Hills Software's INTEGRITY [Gre], are fully preemptible.

We claim that *the perceived need for a fully-preemptible design of a protected RTOS is a myth.*

Specifically, we assert that, for systems employing dual-mode execution and memory protection, a non-preemptible design can achieve *similar worst-case performance* as a fully-preemptible approach, while remaining much simpler (and therefore more trustworthy) and providing better average-case performance.

We support this claim by examining the worst-case execution time (WCET) that can be achieved by a non-preemptible kernel, our seL4 microkernel [KEH$^+$09]. For comparison, we perform a WCET analysis of a system using a mature, fully-preemptible design, the commercial QNX microkernel [QNX]. Our initial study shows that the worst-case interrupt latencies achievable by the two systems are at least similar to those of QNX, suggesting that there is no inherent advantage to the fully-preemptible design that would justify the added complexity and reduced average-case performance.

2 Background

While the traditional RTOS design is generally referred to as *fully preemptible*, this notion has to be taken with a grain of salt: there is always some code in the kernel which needs to execute atomically, and thus cannot be preempted. Atomicity is achieved by disabling interrupts momentarily, which on a multiprocessor is combined with spin locks. Uninterruptible sequences include kernel entry and exit, interrupt dispatching, and accesses to data structures shared between multiple execution contexts.

The alternative design, referred to as *non-preemptible*, disables interrupts while the kernel is executing, resulting in a much simpler kernel design and implementation, as concurrency issues are avoided. The resulting RTOS code is easier to understand, debug and reason about. It also simplifies the process of formally proving the functional correctness of the code, something that is presently considered infeasible for preemptible programs given their high levels of concurrency [KEH$^+$09]. The simplicity also tends to result in better average-case performance, which is the reason this approach has traditionally been taken in L4 microkernels.

RTOS designers, even when targeting high-performance processors typically used for protected-mode systems, aim to achieve worst-case interrupt latencies of the order of tens of microseconds or less. While this allows for a fair amount of computation, it is difficult (if not impossible) to design an practical RTOS where all system calls are so short. However, interrupt latency be reduced by adding *preemption points* in long-running kernel operations. Preemption points respond to any pending interrupts immediately, introducing a limited amount of concurrency. The approach requires that kernel data structures be in consistent states when these points are encountered. Furthermore, the designer must ensure that the interrupted operations are eventually completed, and that high interrupt rates do not impede progress. These requirements complicate and limit the placement of preemption points.

Note that we do not claim that a non-preemptible approach is always the right choice. System calls in monolithic operating systems, such as Windows, Linux or Mac OS, can execute for a long time. Limiting the WCET in such a case would require a very large number of preemption points, resulting in negligible gain over a preemptible approach due to the added complexity. In a well-designed microkernel, however, most kernel operations are very short, and are dominated by entry/exit and context-switching costs. Protected-mode RTOSes are inevitably microkernels, which is why we target this class of system in our study.

At the other extreme, the fully-preemptible design makes sense for a classical, unprotected RTOS, where most interrupts can be handled with minimal switching of state. Such RTOSes can achieve interrupt latencies of up to a few hundred cycles. Hofer et al. have achieved latencies bordering on the limits of the hardware by taking advantage of the CPU's interrupt dispatcher for scheduling [HLSSP09], well below what is feasible in a protected-mode kernel.

The classical RTOS is the best choice in a deeply-embedded system, where a small microcontroller runs relatively simple control software. However, real-time systems are becoming highly complex, running large software stacks, which are hard to debug and assure when using a flat address-space model, increasing the attraction of protected-mode RTOSes. Furthermore, mixed-criticality systems require strong isolation between subsystems, which call for hardware-enforced memory protection.

These developments drive the uptake of protected

RTOSes, and it makes sense to think about the best approach to their design. We have already mentioned the difficulty in reasoning about highly-concurrent code, which results from a fully-preemptible design.

Given the safety-critical nature of many hard real-time systems, trustworthiness of the OS is an issue of growing importance. The recent formal proof of functional correctness of the seL4 microkernel shows that it is a promising platform for building safety-critical systems. This formal verification was only feasible due to the non-preemptible design of seL4 [KEH^{+}09].

On the other hand, a fully-preemptible kernel, where interrupts are only disabled for short pieces of code, is far easier to analyse for the worst-case interrupt latency, which is a possible reason why this approach is generally used in industry. However, as we have recently shown on seL4 [BSC^{+}11], it is feasible to perform a complete and sound WCET analysis of a non-preemptible kernel.

Despite significant effort invested into reducing the WCET of seL4, it is still in the hundreds of microseconds [BSH12], which is at least an order of magnitude larger than what is achievable by preemptible kernels, even protected ones. However, the work on seL4 was subject to the additional constraint that the resulting kernel code can still be functionally verified, something that is presently not achievable at all for preemptible kernels. We are facing a trade-off between assurance and timeliness.

If we are willing to lower our expectation of assurance to a level closer to that of present commercial systems (meaning we appease ourselves with the traditional approaches of testing and code inspection, augmented with model checking) then the WCET of a non-preemptible kernel can be greatly reduced, to the point where it approaches the WCET achievable with the preemptible approach. Even if not formally verified, the resulting non-preemptible kernel is much less complex, and as such easier to assure, than a fully-preemptible one.

3 Approach

We perform our analysis by adapting the tools and techniques developed for our previous work in the timing analysis of seL4 [BSC^{+}11]. Our process involves performing a static analysis to determine the worst-case execution time between two given points in a program, using a model of the hardware to conservatively estimate the timing of cached and pipelined execution (based on a modified version of Chronos [LLMR07]).

We refer the reader to our previous work for further details on our toolchain and method to analyse seL4.

3.1 Analysis targets

A non-preemptible kernel Most L4 kernels use a non-preemptible design. We focused on the seL4 microkernel as representative of such a design, as we already have a comprehensive WCET analysis of the entire microkernel [BSC^{+}11].

As seL4 runs with interrupts disabled during kernel execution, preemption points are used to limit interrupt latency. Therefore, the worst-case interrupt latency of seL4 is given by the sum of the WCETs of the longest non-preemptible path and the interrupt delivery path to a user process. seL4's preemption points have so far been placed within routines that traverse unbounded data structures, and hence are potentially long-running.

In order to achieve good performance, microkernel-based systems rely on very fast IPC operations, typically provided by IPC *fastpaths* [Lie93]. A fastpath usually improves IPC times by orders of magnitude, as it handles only the most commonly executed operations, deferring other operations to the conventional *slowpath*. As adding preemption points in a code path increases its uninterrupted execution time, making the IPC fastpath preemptible would significantly degrade average-case performance. Furthermore, given the very limited functionality of the IPC fastpath, it would be very difficult (if not impossible) to preempt it while still guaranteeing progress.

For these reasons, we left the IPC fastpath non-preemptible, and use the WCET of the fastpath as the target latency for all other non-preemptible code paths in seL4. This is based on our experience with reducing the WCET of seL4 [BSH12]. There we found that it was generally possible to space preemption points almost arbitrarily closely in the longer kernel operations, but in many such cases the re-verification effort would have been too high for the time being. While we have not applied such changes to the kernel yet, we are confident that all code paths can be modified to limit their non-preemptible parts to run for no longer than the WCET of the IPC fastpath.

A fully-preemptible kernel There are several fully-preemptible, general-purpose kernels used in real-time systems today, including QNX's Neutrino [QNX], Wind River's VxWorks [Win] and Green Hills Software's INTEGRITY [Gre].

We analysed the QNX Neutrino microkernel as a representative of the fully-preemptible approach due to its maturity and broad real-world adoption. QNX source was made publicly available in 2008. Although our WCET analysis works on a compiled binary, the availability of source code was important to us in order to help direct the analysis. The manual intervention steps benefit substantially from a detailed knowledge of the program's internals.

We based our analysis on a QNX source snapshot from approximately July 2009, as QNX source code is unfortunately no longer generally available to the public. We performed a cursory comparison between the compiled assembly of our snapshot and the latest QNX Neutrino 6.5.0 binary, and concluded that the intervening changes were insubstantial to our results.

3.2 Analysing QNX

Characterising the interrupt latency of a fully-preemptible kernel requires analysing two groups of code: (1) all regions where interrupts are disabled in the kernel; and (2) the kernel's interrupt dispatch routine which processes incoming interrupts and delivers them to a user-space interrupt handler. The worst-case interrupt latency for the highest-priority interrupt is the sum of these two.

Control-flow graph extraction As QNX was not written with the same emphasis on verifiability as seL4, its construction was less amenable to static analysis. This called for a number of changes to our analysis tools, in particular our control flow graph (CFG) extraction tool, Quoll.

Quoll was initially written with the express purpose of performing a WCET analysis on seL4. This meant that it implemented little more than what was needed, so we had to extend it to support machine instructions that were used in QNX but not in seL4.

We also extended Quoll to allow the user to manually specify the destination of indirect calls to function pointers, which were not present in seL4. Fortunately, these could all be resolved statically.

As the QNX code base is far larger than that of seL4, the initially-generated CFG was prohibitively large and complex. We tackled this by adding features to Quoll to support analysing only specific subsets of the CFG. This was achieved by forcing specific branches to be marked as taken or ignored in this early stage of the analysis. We ensured that our analysis remained sound by ensuring that the subset analysed contained part of the critical path for interrupt delivery.

Our tools are still limited in what they can analyse, and we had to take a few shortcuts. To ensure the validity of our hypothesis, we took, where necessary, an optimistic approach to the analysis of QNX (i.e. we under-estimate its WCET) and a pessimistic one with seL4 (i.e. over-estimation).

For instance, we ignored QNX's support for sporadic scheduling, as this would have significantly complicated our analysis; by ignoring this we underestimate the WCET of QNX. Furthermore, we found that QNX has a suboptimal implementation of a priority queue (a sorted linked list with $\mathcal{O}(n)$ complexity); we ignored this as there is obviously a better $\mathcal{O}(\log n)$ implementation.

Analysing runtime-generated code The QNX Neutrino RTOS follows a generic pattern in its design. For each supported processor architecture (ARM, x86, etc.), QNX provides a single *procnto* binary (modulo options such as instrumentation). QNX also provides generic board support package (BSP) code. System integrators provide their own machine-specific code (for booting, hardware control, etc.) using the QNX BSP code as a base.

A large proportion of the interrupt-handling code depends on the specifics of the machine. Unfortunately, QNX accomplishes this in a way that complicates the static analysis process. At boot time, the interrupt vector code is dynamically generated based on the system's interrupt topology, and is interspersed with integrator-provided code to interface with the interrupt controllers. While this has allowed QNX to ship a single binary that does not need static relocation or other build-time transformations, it also means that the interrupt vector code is not directly available.

We had observed that the generated interrupt vector code in memory did not change after booting and so we were able to simply dump the generated code from a running system. We proceeded with our analysis by re-assembling and linking the dumped code with the rest of the *procnto* image. In this way, we were able to preserve symbols, aiding in the maintenance of our manual annotations.

Event handling Although QNX provides a number of different interrupt delivery methods to userspace, we focus on one mechanism provided by QNX, namely using the `InterruptWait` system call. This mechanism is functionally equivalent to that used in seL4. More importantly, of the different interrupt delivery methods available, it represents a best case and therefore a lower bound on the time required to dispatch an interrupt to userspace code. Other methods, such as signal delivery or thread creation, require significantly more work to process an interrupt, compared with waking an existing blocked thread.

3.3 Improving seL4's WCET

We have previously shown that the WCET of many long-running operations in seL4 can be reduced while preserving verifiability [BSH12]. Here we describe several improvements to seL4 that can reduce the worst-case interrupt latency further, if we drop the requirement for formal verification. This enables optimisations which reduce WCET or allow more liberal use of preemption points.

The longest path through seL4 corresponds to a worst-case IPC operation. It is triggered by sending a full-length message, accompanied by a number of *capability transfers* that are used to delegate access to kernel objects. seL4 uses capabilities [DVH66] for access control, and allows system designers to specify the "address" of a capability in a 32-bit *capability space*, represented internally as a directed graph. Although this provides a large degree of flexibility to system designers, it impacts negatively on real-time performance, as decoding the address of a capability requires traversing up to 32 edges of a directed graph. Within a single kernel invocation, this may occur up to 11 times, generating an immense number of potential cache misses.

Removing capability addressing We can begin improving the WCET for seL4 by forgoing its flexible capability addressing scheme, ensuring that all capabilities can be resolved by traversing only a single level within the capability space. Most RTOSes, including QNX, do not offer anything comparable to the flexible addressing scheme of seL4. At worst this makes seL4's addressing on par with other RTOSes, where a single level of indirection is typically used to address kernel objects.

To accomplish this, we need not modify seL4; instead, system designers can use seL4's built-in authority model to prevent untrusted code from creating more than one level of addressing. This allows us to reduce the number of potential cache misses caused by address lookups by a factor of 32.

Atomic send-receive seL4's IPC path latency can also be improved by making certain IPCs preemptible. As mentioned earlier, we do not wish to make the IPC fastpath preemptible as it would impact on system performance. However, the IPC slowpath can be made preemptible in one specific case—the *ReplyWait* operation, which atomically performs the functionality of two operations commonly called together: *Reply* and *Wait*. The IPC fastpath relies on the atomicity of these operations in order to attain a significant performance benefit. However, if an IPC operation falls back to the slowpath, there is a negligible penalty in adding a preemption point between the send and receive phases.

Global kernel mapping copy The seL4 kernel is mapped into virtual memory in the top 256 MiB of every address space on the system. This mapping is represented by 1 KiB of data in every page directory, which must be copied each time a new page directory is created. The process of copying these mappings is currently non-preemptible. Two possible solutions are to (a) simply add preemption points to the copy operation, or (b) move the entire seL4 kernel into its own 32-bit address space. The latter solution simplifies the kernel, avoiding the address space copy entirely. However, it also impacts negatively on overall performance, as a context switch is then required for every kernel entry and exit. Although more complex to implement, we assume the first approach is used in our analysis.

Cache pinning Although CPUs have increased in speed significantly in the past 30 years, memory has not kept the same pace. As a result, operations which are inherently memory-intensive have not seen a great improvement from increased CPU speed. The increased disparity between CPU and memory speeds have pushed hardware manufacturers to include one or more levels of caches even in embedded systems.

Code region	QNX WCET	seL4 WCET
Longest D-I region	>4 441	∼22 000
Interrupt to userspace	>17 413	∼10 000
Total:	>21 854	∼32 000

Table 1: The upper-bound WCET of seL4, and a subset of code paths in QNX which directly affect interrupt latency, in cycles.

These caches can be accessed significantly faster than main memory. For example, our test platform has two levels of caches—the first can be accessed in a single cycle and the second in 26 cycles, compared with external memory latency of 96 cycles. However, these fast caches are typically much smaller than main memory.

As microkernels are intended to provide the bare minimum functionality for ensuring security of a system, they are typically very small. An seL4 microkernel binary can fit into 36 KiB. The L2 cache on our platform is 128 KiB, which means that it would be possible to fit almost the entire kernel code into 1/4 of the L2 cache. Our analysis therefore assumes the kernel is pinned in the L2 cache, reducing the latency of L1 instruction cache misses by 73 %.

4 Results

Using our WCET analysis tools, we computed the WCET of a subset of the QNX interrupt handling paths. There are other paths through QNX which we know to have a longer interrupt delivery time, however they proved more difficult to analyse. Thus the results presented for QNX are not absolute worst cases but instead lower bounds on the worst case, which we believe to be suitably representative of a fully-preemptible kernel.

Table 1 shows the results of our analysis of both QNX and seL4. On both QNX and seL4, the worst-case interrupt latency is the sum of both the longest region with interrupts disabled, and the time to dispatch an interrupt to a user thread. For QNX, this would lead to a minimum worst case of about 22 k cycles. The longest disabled-interrupt (D-I) region actually corresponds to a portion of the interrupt dispatch routine which must execute with interrupts off.

For comparison, we investigated the limits of WCET of a mostly non-preemptible kernel, our modified seL4. We estimated the WCET of the longest disabled-interrupt region by adapting our previous analysis [BSH12] to model the effects of the modifications described in Section 3.3. The interrupt dispatch time is the worst-case execution time of the interrupt delivery path in seL4.

The totals in Table 1 show the worst-case interrupt latencies of a representative fully-preemptible kernel and non-preemptible kernel as roughly 22 k cycles and 32 k cycles, respectively. These results suggest that the worst-case interrupt latency achievable by a well-designed non-preemptible kernel is no more than a factor of 1.5 from that of a fully-preemptible kernel. If we consider that QNX's interrupt dispatch to userspace could be improved to be on par with our estimate for seL4, the non-preemptible kernel is still only a factor of 2.2 worse-off than a fully-preemptible one.

5 Conclusion

We have explored the differences between the worst-case interrupt latencies of (mostly) non-preemptible and (almost) fully-preemptible kernels. Low worst-case interrupt latencies are one crucial aspect of building mixed-criticality systems supporting hard real-time applications.

We have presented strong evidence to suggest that a non-preemptible kernel design can provide worst-case interrupt latencies competitive with that of a fully-preemptible kernel. Although these are preliminary results, it is promising for system builders who are seeking to create trustworthy systems with enhanced levels of assurance enabled by the relative ease of reasoning about non-preemptible systems.

Our future work aims to implement the modifications to seL4 to reduce the interrupt response time to approximately 30 k cycles. We also plan to examine more deeply the reasons for the interrupt latencies applicable to both types of systems, and explore the feasibility of reducing worst-case interrupt latencies even further.

Acknowledgements

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

[BSC$^+$11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *32nd RTSS*, Vienna, Austria, Nov 2011.

[BSH12] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *7th EuroSys Conf.*, pages 323–336, Bern, Switzerland, Apr 2012.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[Gre] Green Hills Software. INTEGRITY real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[HLSSP09] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *30th RTSS*, 2009.

[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.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Asheville, NC, USA, Dec 1993.

[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), Dec 2007.

[QNX] QNX. Operating systems. http://www.qnx.com/products/neutrino-rtos/.

[Win] Wind River. Wind River VxWorks RTOS. http://windriver.com/products/vxworks/.