Improving Interrupt Response Time in a Verifiable Protected Microkernel

Bernard Blackham, Yao Shi and Gernot Heiser
NICTA and University of New South Wales, Sydney, Australia
First.Last@nicta.com.au

Abstract

Many real-time operating systems (RTOSes) offer very small interrupt latencies, in the order of tens or hundreds of cycles. They achieve this by making the RTOS kernel fully preemptible, permitting interrupts at almost any point in execution except for some small critical sections. One drawback of this approach is that it is difficult to reason about or formally model the kernel's behavior for verification, especially when written in a low-level language such as C.

An alternate model for an RTOS kernel is to permit interrupts at specific preemption points only. This controls the possible interleavings and enables the use of techniques such as formal verification or model checking. Although this model cannot (yet) obtain the small interrupt latencies achievable with a fully-preemptible kernel, it can still achieve worst-case latencies in the range of 10,000s to 100,000s of cycles. As modern embedded CPUs enter the 1 GHz range, such latencies become acceptable for more applications, particularly when they come with the additional benefit of simplicity and formal models. This is particularly attractive for protected multitasking microkernels, where the (inherently non-preemptible) kernel entry and exit costs dominate the latencies of many system calls.

This paper explores how to reduce the worst-case interrupt latency in a (mostly) non-preemptible protected kernel, and still maintain the ability to apply formal methods for analysis. We use the formally-verified seL4 microkernel as a case study and demonstrate that it is possible to achieve reasonable response-time guarantees. By combining short predictable interrupt latencies with formal verification, a design such as seL4's creates a compelling platform for building mixed-criticality real-time systems.

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design—Real-time systems and embedded systems; D.4.5 [*Operating Systems*]: Reliability—Verification; D.4.8 [*Operating Systems*]: Performance—Modeling and prediction

General Terms Design, Performance, Reliability

Keywords Microkernels, worst-case execution time, hard real-time systems, trusted systems, formal verification

1. Introduction

Hard real-time systems are regularly deployed in critical environments such as cars, aircraft and medical devices. These systems demand both functional correctness as well as precise timing guarantees. A failure to meet either functional or timing requirements may result in catastrophic consequences. As manufacturers strive to gain a competitive advantage by combining both critical and convenience functionality, the overall complexity of devices has increased. Maintaining the stringent demands on safety and reliability of these systems is paramount.

One approach to improve reliability is to physically separate critical subsystems onto separate processors, with minimal interference from other subsystems. Whilst this approach has its benefits, it does not scale to the more complex devices being created today, which would require tens or possibly hundreds of processors, each servicing different subsystems. The added weight, cost and power consumption of these processors are severe drawbacks to this approach. It also provides poor support for (controlled) communication between the otherwise isolated subsystems; in an embedded system, where all subsystems cooperate to achieve the overall system mission, this can result in performance degradation and increased energy use.

An alternative approach is to consolidate multiple subsystems onto a single processor, and use a trustworthy supervisor to provide functional and temporal isolation [Mehnert 2002]. The supervisor also provides controlled communication between selected components. This approach is represented in Figure 1. The supervisor may be a microkernel, hypervisor or protected real-time operating system (RTOS),

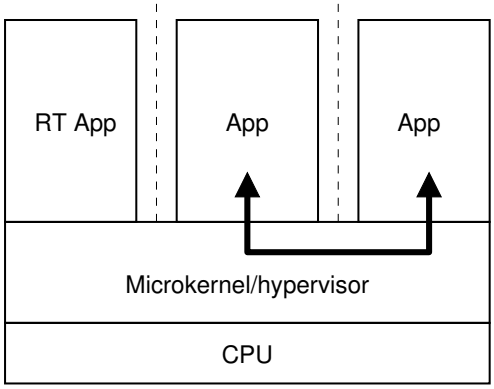

Figure 1. A trustworthy microkernel or hypervisor can be used to provide functional and temporal isolation between distinct components executing on a single processor.

and should have a small trusted code base. The supervisor must be able to provide the required temporal guarantees such that all real-time deadlines can be met, and also protect components from each other.

seL4 [Klein 2009b] is the world's first formally-verified general-purpose operating system kernel. seL4 offers a machine-checked proof that the C implementation adheres to a formal specification of the kernel's behaviour. seL4 is a third-generation microkernel broadly based on the concepts of L4 [Liedtke 1995]. It provides virtual address spaces, threads, inter-process communication and capabilities [Dennis 1966] for managing authority.

We have previously shown that the design of seL4 enables a complete and sound analysis of worst-case interrupt response time [Blackham 2011a]. Combining this with formal verification makes seL4 a compelling platform for building systems with critical hard real-time functionality co-located with other real-time or best-effort components. The size and structure of seL4's code base have also been key ingredients to the successful formal verification of its functional behaviour.

Like most of its predecessor L4 kernels, seL4 disables interrupts throughout kernel execution, except at specific preemption points. This was originally done to optimise average-case performance, at the expense of increased interrupt latency. However, it was also essential to making the formal verification of the functional behaviour of seL4 tractable. As a consequence, seL4 is unable to achieve the very short interrupt latencies of a fully-preemptible kernel (within hundreds of cycles). However, with embedded processor speeds of 1 GHz or higher becoming common-place, many applications on these platforms can tolerate latencies in the tens of thousands, or even hundreds of thousands of cycles. We show that seL4 is able to guarantee worst-case latencies of this magnitude, whilst retaining the ability to offer strong functional guarantees.

Mixed-criticality systems with reasonable latency requirements are therefore well-suited to this model. Some examples include medical implants, or automotive systems for braking or ABS. These devices demand functional and temporal correctness but their required latencies are in the order of milliseconds.

In this paper, we explore the limits of the response time of a verified protected microkernel such as seL4. Previous analyses have been instrumental in the development of seL4's real-time properties [Blackham 2011a;b]. We describe the bottlenecks encountered in achieving suitable real-time performance and the challenges associated with overcoming them. Although re-verification of these changes is ongoing work, it is largely mechanical and similar in nature to other proof maintenance activities over the past two years.

2. Avoiding Preemption in the Kernel

OS kernels are typically developed using either a process-based model or an event-based model supporting multi-threading. In the process-based model, each thread is allocated a dedicated kernel stack. This kernel stack is used for handling requests on behalf of the thread, and implicitly stores the state of any kernel operation in progress through the contents of the call stack and local variables stored on the stack.

A process-based kernel lends itself to being made fully preemptible. With some defensive coding, an interrupt can be permitted almost anywhere in the kernel, allowing for very short interrupt response times.

An event-based kernel uses a single kernel stack to handle requests for all threads on the system. This reduces memory consumption significantly, but changes the way in which preemption can be handled. In the general case, this stack cannot be shared amongst multiple threads as doing so could result in deadlock due to stack blocking. Scheduling disciplines such as the stack resource policy [Baker 1991] are able to re-use stacks and avoid this. However, seL4 is not bound to a specific scheduling discipline – currently, a fixed-priority preemptive scheduler is used, and alternatives are in development.

To preempt threads running on a single kernel stack, preemption points are manually inserted into seL4 to detect and handle pending interrupts. If an interrupt is pending, the state of the current operation must be explicitly saved.

Continuations can be used in an event-based kernel to efficiently represent the saved state of a kernel operation that has blocked or been preempted [Draves 1991]. A continuation specifies (a) the function that a thread should execute when it next runs; and (b) a structure containing any necessary saved state. Using continuations allows a thread to discard its stack whilst it is blocked or preempted.

L4 kernels have traditionally been process-based but not fully preemptible[1]. The process-based kernel had previously been claimed to be more efficient in the presence of frequent context switching [Liedtke 1993], leading to this favoured design. However, experiments on modern ARM hardware have shown a negligible difference between the two models [Warton 2005]. Furthermore, in a microkernel-based system, kernel execution tends to be dominated by fast interprocess communication (IPC) operations; there is little benefit in making the microkernel fully-preemptible, as long as the worst-case latencies of the longer-running operations are kept reasonable through well-placed preemption points.

2.1 Design of seL4

As seL4 is event-based, context switches involve no stack manipulation. This simplifies formal verification, as the seL4 code has a standard procedural control flow like any regular C program. The most common operations in seL4 (IPC) are designed to be short. However, object creation and deletion operations are necessarily longer, as they require iterating over potentially large regions of memory and manipulating complex data structures.

In seL4, a preempted operation is effectively a restartable system call. Interrupts are disabled in hardware during kernel execution, and handled when encountering a preemption point or upon returning to the user. At a preemption point, any necessary state for the preempted operation is saved as seL4 returns up the call stack. The system is left in a state where simply re-executing the original system call will continue the operation.

This is in contrast to using continuations for each thread and maintaining the state of the operation in progress. By not using continuations, the amount of code required is reduced, as the safety checks needed for resuming a preempted operation are similar to those required for starting the operation from scratch; kernel re-entry automatically re-establishes the required invariants. It also simplifies reasoning about the kernel's correctness: it is not necessary to reason about what state a preempted thread is in and whether it was performing an operation, instead it suffices to reason about the state of objects in the system.

This design leads to a small amount of duplicated effort, as the system call must be decoded again each time a preempted operation is resumed. However, the code paths taken are likely to be hot in the CPU's caches. Work on the Fluke kernel demonstrated that these overheads are negligible [Ford 1999].

Incremental Consistency A noteworthy design pattern in seL4 is an idea we call *incremental consistency*: large composite objects are composed of individual components that can be added or deleted one-at-a-time. Specifically, there is always a constant-time operation that will partially construct or deconstruct a large composite object and still maintain a coherent system. In seL4, this is relevant to objects such as address spaces and book-keeping data structures, which commonly pose significant issues in deletion paths.

Although a simple concept, it is not trivial to maintain in a system with complex data structures and intricate dependencies between them. Ensuring that composite objects can be incrementally created and destroyed has benefits for verification and also naturally lends itself to preemption.

2.2 Proof Invariants of seL4

In seL4, consistency of the kernel is defined by a set of formalised invariants on the kernel's state and in turn all objects in the kernel. There are in fact hundreds of invariants and lemmas that are maintained across all seL4 operations. These include:

- *well-formed data structures*: structures such as linked lists are well-formed – i.e., there are no circular links and all back-pointers point to the correct node in doubly-linked lists;
- *object alignment*: all objects in seL4 are aligned to their size, and do not overlap in memory with any other objects;
- *algorithmic invariants*: some optimisations in seL4 depend on specific properties being true, allowing redundant checks to be eliminated;
- *book-keeping invariants*: seL4 maintains a complex data-structure that stores information about what objects exist on the system and who has access to them. The integrity of seL4 depends on the consistency of this information.

The formally-verified seL4 code base proves that all kernel operations will maintain all of the given invariants. Any modifications to seL4 require proving that these invariants still hold, in addition to proving that the code still correctly implements the specification of the kernel. Therefore, for each preemption point that we add to seL4, we must correspondingly update the proof in order to maintain these invariants.

In some cases, it is not possible to maintain all invariants, in which case the invariant may be replaced by a weaker statement. The weakened invariant must be sufficient to satisfy the remainder of the proof over the whole kernel. If an aspect of the proof fails with the weakened invariant, this generally suggests that a bug has been introduced and that extra safety checks may be required in the code.

3. Areas of Improvement

In this section, we look at some of the long-running operations in seL4 and examine how to either add suitable preemption points or replace the operations with better algorithms. Most of these operations, whilst presented in the con-

[1] There are some exceptions: the L4-based Fiasco kernel [Hohmuth 2002] is process-based and also preemptible; both OKL4 [Heiser 2010] and seL4 use a single kernel stack and are therefore not fully preemptible.

```
thread_t chooseThread() {
  foreach (prio in priorities) {
    foreach (thread in runQueue[prio]) {
      if (isRunnable(thread))
        return thread;
      else
        schedDequeue(thread);
    }
  }
  return idleThread;
}
```

Figure 2. Pseudo-code of scheduler implementing lazy scheduling.

```
thread_t chooseThread() {
  foreach (prio in priorities) {
    thread = runQueue[prio].head;
    if (thread != NULL)
      return thread;
  }
  return idleThread;
}
```

Figure 3. Pseudo-code of scheduler without lazy scheduling.

text of seL4, are typical of any OS kernel providing abstractions such as protected address spaces, threads and IPC.

There are some operations that may be found in other OS kernels which are not present in seL4. One example is the absence of any memory allocation routines. Almost all allocation policies are delegated to userspace; seL4 provides only the mechanisms required to enforce policies and ensure they are safe (e.g. checking that objects do not overlap) [Elkaduwe 2007]. This design decision removes much of the complexity of a typical allocator from seL4 as well as some potentially long-running operations.

3.1 Removal of Lazy Scheduling

The original version of seL4 featured an optimisation known as *lazy scheduling*. Lazy scheduling attempts to minimise the manipulation of scheduling queues on the critical path of IPC operations [Liedtke 1993], and has traditionally been used in almost all L4 kernels (Fiasco is an exception). It is based on the observation that in L4's synchronous IPC model, threads frequently block while sending a message to another thread, but in many cases the other thread replies quickly. Multiple such ping-pongs can happen on a single time slice, leading to repeated de-queueing and re-queueing of the same thread in the run queue.

Lazy scheduling leaves a thread in the run queue when it executes a blocking IPC operation. When the scheduler is next invoked, it dequeues all threads which are still blocked. Pseudo-code for a scheduler implementing lazy scheduling is shown in Figure 2.

Lazy scheduling can lead to pathological cases where the scheduler must dequeue a large number of blocked threads (theoretically only limited by the amount of memory available for thread control blocks), which obviously leads to horrible worst-case performance. As the scheduler is responsible for determining which thread to run next, it is not feasible or even meaningful to add a preemption point to this potentially long-running operation.

We therefore had to change the scheduling model so that only runnable threads existed on the run queue. In order to maintain the benefits of lazy scheduling, we use a different scheduling trick, (internally known as "Benno scheduling", after the engineer who first implemented it in an earlier version of L4): when a thread is unblocked by an IPC operation and, according to its priority, it is able to execute immediately, we switch directly to it and do not place it into the run queue (as it may block again very soon). The run queue's consistency can be re-established at preemption time, when the preempted thread must be entered in the run queue if it is not already there. This has the same best-case performance as lazy scheduling, but maintains good worst-case performance, as only a single thread (the presently running one) may have to be enqueued lazily.

In this model the implementation of the scheduler is simplified, as it now just chooses the first thread of highest priority, as demonstrated in the pseudo-code listing in Figure 3. There is an existing invariant in the kernel that all runnable threads on the system are either on the run queue or currently executing. This is sufficient for lazy scheduling, but Benno scheduling obviously requires an additional invariant which must be maintained throughout the kernel: that all threads on the scheduler's run queue must be in the runnable state.

This seemingly simple C code change impacts the proof in all functions that alter a thread's state, or modifies the scheduler's run queue. The invariant must be proven true when any thread ceases to be runnable and when any thread is placed onto the run queue.

3.2 Scheduler Bitmaps

We added one more optimisation to the scheduler: a bitmap representing the priorities that contain runnable threads. We make use of ARM's *count leading zeroes* (CLZ) instruction which finds the highest set bit in a 32-bit word, and executes in a single cycle. seL4 supports 256 thread priorities, which we represent using a two-level bitmap. The 256 priorities are divided into 8 "buckets" of 32 priorities each. The top-level bitmap contains 8 bits each representing the existence of runnable threads in any of the 32 priorities within a bucket. Each bucket has a 32-bit word with each bit representing one of the 32 priorities. Using two loads and two CLZ instructions, we can find the highest runnable priority very

efficiently, and have thus removed the loop from Figure 3 altogether.

This optimisation technique is commonly found in OS schedulers and has reduced the WCET of seL4 in several cases. However, it introduces yet another invariant to be proven: that the scheduler's bitmap precisely reflects the state of the run queues. As this is an incremental change to the existing scheduler, the re-verification effort is significantly lowered.

3.3 Aborting IPC Operations

Threads in seL4 do not communicate directly with each other; they instead communicate via *endpoints* which act as communication channels between threads. Multiple threads may send or receive messages through an endpoint. Each endpoint maintains a linked list of all threads waiting to send or receive a message. Naturally, this list can grow to an arbitrary length (limited by the number of threads in the system, which is limited by the amount of physical memory that can be used for threads, which in turn is theoretically limited by the size of free physical memory after the kernel boots). The length of this list is not an issue for most operations, as they can manipulate the list in constant time.

The only exception is the operation to delete an endpoint. Deletion must iterate over and dequeue a potentially large number of threads. There is an obvious preemption point in this operation: after each thread is dequeued. This intermediate step is fortunately consistent with all existing invariants, even if the thread performing the deletion operation is itself deleted. Forward progress is ensured by deactivating the endpoint at the beginning of delete operations, so threads (including those just dequeued) cannot attempt another IPC operation on the same endpoint.

The preemption point here is obvious because it is a direct result of the incremental consistency design pattern in seL4. As a result, the impact of these changes on the proof are minimal.

3.4 Aborting Badged IPC Operations

A related real-time challenge in seL4 is the aborting of *badged* IPC operations. Badges are unforgeable tokens (represented as an integer) that server processes may assign to clients. When a client sends a message to the server using a badge, the server can be assured of the authenticity of the client. A server may revoke a specific badge, so that it can ensure that no existing clients have access to that badge. Once revoked, the server may re-issue the badge to a different client, preserving guarantees of authenticity.

In order to revoke a badge, seL4 must first prevent any thread from starting a new IPC operation using the badge, and second ensure that any pending IPC operations using the badge are aborted. It is this second operation which requires a compromise between execution time, memory consumption and ease of verification. The choice of data structure used to store the set of pending IPC operations (clients and their badges) has a significant impact on all three factors.

For example, a balanced binary-tree structure has very good worst-case and average-case execution time, but requires more work on the verification effort; the invariants involved in self-balancing binary tree structures are more tedious than linear data structures. A hash-table-based data structure may be easier to verify, and has good average-case performance, but raises challenging memory allocation issues in seL4, where memory allocation is handled outside the kernel. It also does not improve worst-case execution time, as a determined adversary could potentially force hash collisions.

Instead, seL4 uses a simple linked list containing the list of waiting threads and their associated badges, as described in Section 3.3. Enqueuing and dequeuing threads are simple O(1) operations. In order to remove all entries with a specific badge, seL4 must iterate over the list; this is a potentially unbounded operation, and so we require a preemption point. Unlike the simple deletion case where we simply restart from the beginning of the list, here we additionally need to store four pieces of information:

1. at what point within the list the operation was preempted, so that we can avoid repeating work and ensure forward progress;
2. a pointer to the last item in the list when the operation commenced, so that new waiting clients do not affect the execution time of the original operation;
3. the badge which is currently being removed from the list, so that if a badge removal operation is preempted and a second operation is started, the first operation can be completed before starting the new one;
4. a pointer to the thread that was performing the badge removal operation when preempted, so that if another thread needs to continue its operation, it can indicate to the original thread that its operation has been completed.

With all this information, we are able to achieve our goal of incremental consistency. The above information is associated with the endpoint object rather than the preempted thread (as would be done in a continuation). In doing so, we can reason simply about the state of objects in our invariants, rather than the state of any preempted thread.

Note that although the preemption point bounds interrupt latency, this approach gives a longer than desirable execution time for the badged abort operation, as every waiting thread must be iterated over, rather than only threads waiting for a specific badge. This has not yet been an issue in real systems, however, should it cause problems then we may replace it with an alternative such as a self-balancing binary tree data structure and undertake the extra verification effort required.

3.5 Object Creation

When objects, such as threads, page tables or memory frames, are created on behalf of the user, their contents must

be cleared and/or initialised in order to avoid information leakage. Clearing an object may be a long-running operation, as some kernel objects are megabytes in size (e.g. large memory frames on ARM can be up to 16 MiB; capability tables for managing authority can be of arbitrary size).

The code to clear an object was previously deep inside the object creation path, and replicated for each type of object. Additionally, the code updated some of the kernel's state before objects were cleared, and the rest of the kernel's state after objects were cleared. Adding a preemption point in the middle of clearing an object would therefore leave the kernel in an inconsistent state.

To make clearing of objects preemptible, seL4 required significant restructuring of the object creation paths. We chose to clear out the contents of all objects prior to any other kernel state being modified. The progress of this clearing is stored within the object itself. As clearing is the only long-running aspect of these operations, the remainder of the creation code that manipulates the state of the kernel (e.g. updating the kernel's book-keeping data structures) can be performed in one short, atomic pass.

Page directories (top-level page tables) however pose an added complication. The kernel reserves the top 256 MiB of virtual address space for itself, and is mapped into all address spaces. When a new page directory is created, the kernel mappings for this region must be copied in. This copy operation is embedded deep within the creation path of page directories. There is also an seL4 invariant specifying that all page directories will contain these global mappings – an invariant that must be maintained upon exiting the kernel.

Preempting the global-mapping copy poses significant (though not insurmountable) challenges for verification. Fortunately, on the ARMv6 and ARMv7 architectures, only 1 KiB of the page table needs to be updated. We measured the time taken to copy 1 KiB of memory on our target platform (described in Section 5.1) to be around 20 μs. We decided that 20 μs would be a tolerable latency (for now), as there were other long-latency issues to tackle first. Therefore we made all other block copy and clearing operations in seL4 preempt at multiples of 1 KiB, as smaller multiples would not improve the worst-case interrupt latency until the global-mapping copy is made preemptible.

3.6 Address Space Management

In a virtual address space, physical memory frames are mapped into page tables or page directories, creating a mapping from virtual address to physical address. In order to support operations on individual frames, seL4 must additionally maintain the inverse information: which address space(s) a frame has been mapped into, and at which address.

In seL4, all objects are represented by one or more capabilities, or *caps*, which encapsulate metadata about the object such as access rights and mapping information. Capabilities form the basic unit of object management and access control in seL4 systems, and a typical system may have tens or hundreds of thousands of caps. As such, the design of seL4 endeavours to keep caps small to minimise memory overhead. seL4 caps are 16 bytes in size: 8 bytes are used for pointers to maintain their position in a "derivation tree", and the other 8 bytes are used for object-specific purposes.

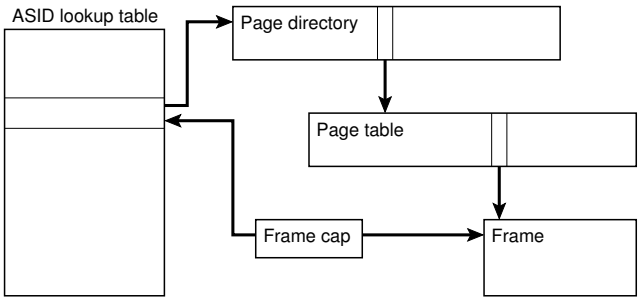

Figure 4. Virtual address spaces managed using ASIDs. Each arrow denotes a reference stored in one object to another.

8 bytes of extra information suffices for almost all objects, however caps to physical memory frames are an exception. To support seL4's object deletion model, frames are required to store their physical address, the virtual address at which they are mapped, and the address space into which they are mapped. This, along with some extra bits of metadata, exceeds the 8-byte limit inside a cap.

In order to squeeze this information into 8 bytes, the original seL4 design uses a lookup table to map from an 18-bit index to an actual address space. The index is called an *address space identifier*, or *ASID*, and is small enough to be stored inside a frame cap. The lookup table is stored as a sparse 2-level data structure in order to minimise space usage, with each second level (*ASID pool*) providing entries for 1024 address spaces. The objects and references required for using ASIDs are shown in Figure 4.

Using ASIDs offered the additional benefit of enabling dangling references to safely exist. If frame caps were to store a reference to the address space itself, then when the address space is deleted, all frame caps referring to it would need to be updated to purge this reference. By instead indirecting through the ASID table, the references from each frame cap, whilst stale, are harmless. Any time the ASID stored in a frame cap is used, it can be simply checked that the mapping in the address space (if any still exist) agrees with the frame cap. As a result, deleting an address space in this design simply involves: (a) removing one entry from the ASID lookup table, and (b) invalidating any TLB entries from the address space.

However, the use of ASIDs poses issues for interrupt latency in other areas, such as locating a free ASID to allocate and deleting an ASID pool. Locating a free ASID is difficult to make preemptible – in the common case, a free ASID would be found immediately, but a pathological case may re-

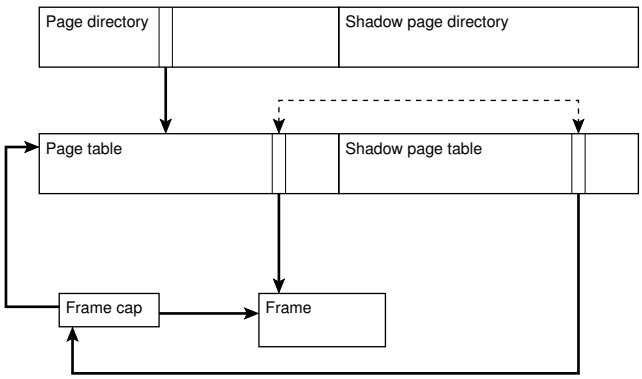

Figure 5. Virtual address spaces managed using shadow page tables. As in Figure 4, each arrow denotes a reference stored in one object to another. The dotted line shows an implicit link by virtue of the shadow being adjacent to the page table itself.

quire searching over 1024 possible ASIDs. Similarly, deleting an ASID pool requires iterating over up to 1024 address spaces. Whilst we could play with the size of these pools to minimise latency, the allocation and deallocation routines are inherently difficult to preempt, and so we decided to seek an alternative to ASIDs.

By removing ASIDs, we needed some other method to prevent dangling references within frame caps. In particular, we needed to store a back-pointer from a virtual address mapping to the frame cap used for the mapping. We chose to store these pointers in a *shadow page table*, mapping from virtual address to frame cap (as opposed to mapping to the frame itself). This effectively doubles the amount of space required for each page table and page directory. We store this data adjacent to the page table in order to facilitate fast lookup from a given page table entry, as shown in Figure 5. Now, all mapping and unmapping operations, along with address space deletion must eagerly update all back-pointers to avoid any dangling references.

This design removes the ability to perform lazy deletion of address spaces, but resolves many of the latency issues surrounding the management of ASID pools. We needed to insert preemption points in the code to delete address spaces, however this is trivial to do and can be done without violating any invariants on the kernel's state. The natural preemption point in the deletion path is to preempt after unmapping each entry in a page table or page directory. To avoid repeating unnecessary work, we also store the index of the lowest mapped entry in the page table and only resume the operation from that point.

We observe that this preemption point is again a direct result of the shadow page table design adhering to the incremental consistency design pattern.

Memory Overhead of Shadow Page Tables The space overhead of shadow page tables might be considered detrimental on some systems. We can compare it to an alternative solution, where we utilise a frame table to maintain bookkeeping information about every physical page on the system. This is the approach used by many other operating systems, including Linux.

The frame table incurs a fixed memory overhead and removes the need for shadows. In its simplest form, without support for sharing pages between address spaces (which the shadow page table solution does support), a frame table would require a single pointer to track each frame cap created. On a 32-bit system with 256 MiB of physical memory and 4 KiB frames, the frame table would occupy 256 KiB of memory. On ARMv6 platforms, page directories are 16 KiB and page tables are 1 KiB. A densely-packed page directory covering 256 MiB of virtual address space would use an extra 256 KiB in shadow page tables, and an extra 16 KiB per address space in page directories.

The shadow page table approach is not significantly less space efficient than using a frame table, except when the system uses sparsely-populated page tables. However, this incurs memory overhead already as the page tables themselves are underutilised, as well as the shadows.

4. Cache Pinning

In order to achieve faster interrupt delivery and tighter bounds on the worst-case execution time (WCET) of seL4, we modified seL4 to pin specific cache lines into the L1 caches so that these cache lines would not be evicted. We selected the interrupt delivery path, along with some commonly accessed memory regions to be permanently pinned into the instruction and data caches. The specific lines to pin were chosen based on execution traces of both a typical interrupt delivery, and some worst-case interrupt delivery paths.

As the cache on our target platform (described in Section 5.1) supports locking one or more complete cache ways, we can choose to lock $1/4$, $1/2$ or $3/4$ of the contents of the cache. We selected as much as would fit into $1/4$ of the cache, without resorting to code placement optimisations. A total of 118 instruction cache lines were pinned, along with the first 256 bytes of stack memory and some key data regions.

The benefit of cache pinning on worst-case execution time is shown in Table 1. On the interrupt path, where the pinned cache lines have the greatest benefit, the worst-case execution time is almost halved. On other paths, the gain is less significant but still beneficial.

Of course, these benefits do not come for free; as a portion of the cache has been partitioned for specific code paths, the remainder of the system has less cache for general usage. A system with hard real-time requirements would also need to ensure that all code and data used for deadline-critical tasks are pinned into the cache. Methods to optimally select cache lines to pin for periodic hard real-time task sets have been the subject of previous research [Campoy 2001, Puaut 2002].

Event handler	Without pinning	With pinning	% gain
System call	421.6 μs	378.0 μs	10%
Undefined instruction	70.4 μs	48.8 μs	30%
Page fault	69.0 μs	50.1 μs	27%
Interrupt	36.2 μs	19.5 μs	46%

Table 1. Improvement in computed worst-case latency by pinning frequently used cache lines into the L1 cache.

Our platform has a 128 KiB unified L2 cache with a hit access latency of 26 cycles (compared with external memory latency of 96 cycles). Our compiled seL4 binary is 36 KiB, and so it would be possible to lock the entire seL4 microkernel into the L2 cache. Doing so would drastically reduce execution time even further, but we have not yet adapted our analysis tools to support this.

5. Analysis Method

After making the changes outlined above, we analysed seL4 to compute a new safe upper bound on its interrupt latency. The analysis was performed on a compiled binary of the kernel and finds the longest paths through the microkernel using a model of the hardware. We also evaluate the overestimation of the analysis by executing the longest paths on real hardware.

5.1 Evaluation Platform

seL4 can run on a variety of ARM-based CPUs, including processors such as the ARM Cortex-A8 which can be clocked at over 1 GHz. However, we were unable to obtain a recent ARM processor (e.g. using the ARMv7 architecture) which also supported cache pinning. In order to gain the benefits of cache pinning, we ran our experiments on a somewhat older processor, the Freescale i.MX31, on a KZM evaluation board. The i.MX31 contains an ARM1136 CPU core with an 8-stage pipeline and is clocked at 532 MHz.

The CPU has split L1 instruction and data caches, each 16 KiB in size and 4-way set-associative. These caches support either round-robin or pseudo-random replacement policies. The caches also provide the ability to select a subset of the four ways for cache replacement, effectively allowing some cache lines to be permanently pinned. Alternately, the caches may also be used as tightly-coupled memory (TCM), providing a region of memory which is guaranteed to accessible in a single cycle.

As our analysis tools do not yet support round-robin replacement (and pseudo-random is not feasible to analyse), we analysed the caches as if they were a direct-mapped cache of the size of one way (4 KiB). This is a pessimistic but sound approximation of the cache's behaviour, as the most recently accessed cache line in any cache set is guaranteed to reside in the cache when next accessed.

The i.MX31 also provides a unified 128 KiB L2 cache which is 8-way set-associative. The KZM board provides 128 MiB of RAM with an access latency of 60 cycles when the L2 cache is disabled, or 96 cycles when the L2 cache is enabled. Due to this significant disparity in memory latency, we analysed the kernel both with the L2 cache enabled and with it disabled.

We disabled the branch predictors of the ARM1136 CPU both on hardware used for measurements and in the static analysis itself, as our analysis tools do not yet model them. Interestingly, using the branch predictor increases the worst-case latency of a branch: with branch predictors enabled, branches on the ARM1136 vary between 0 and 7 cycles, depending on the type of branch and whether or not it is predicted correctly. With the branch predictor disabled, all branches execute in a constant 5 cycles.

The effect of disabling these features on execution time is quantified in Section 6.4.

5.2 Static Analysis for Worst-Case Execution Time

Our analysis to compute the interrupt response time of seL4 is based upon our previous work [Blackham 2011a], to which we refer the reader for more background on the tools used. This section summarises the analysis method and highlights improvements over the previous analysis technique.

We use Chronos 4.2 [Li 2007] to compute the worst-case execution time of seL4 and evaluate our improvements to seL4. We had previously modified Chronos to analyse binaries on the ARMv7 architecture using a model of the Cortex-A8 pipeline. For this analysis we adapted the pipeline model to support the ARM1136 CPU on our target hardware. ARM's documentation of the ARM1136 describes the performance characteristics of the pipeline in detail [ARM 2005].

To analyse the interrupt response time of seL4, we computed upper bounds on the worst-case execution time of all paths through seL4 where interrupts cannot be immediately serviced. These paths begin at one of the kernel's exception vectors: system calls, page faults, undefined instructions or interrupts. A path through the kernel ends when either (a) control is passed back to user and interrupts are re-enabled; or (b) at the start of the kernel's interrupt handler (a preempted kernel operation will return up the call stack and then call the kernel's interrupt handler).

After extracting the control flow graph of the kernel, loops and loop nests are automatically identified. We annotate the control flow graph with the upper bound on the number of iterations of all loops. In this analysis, some of these upper bounds are computed automatically (described further in Section 5.3).

The analysis virtually inlines all functions so that function calls in the control flow graph are transformed into simple branches. This enables the static analysis to be aware of the calling context of a function that is called from multiple call sites; the processor's cache will often be in wildly different

states depending on the execution history. Unfortunately, this virtual inlining also leads to significant overestimation as described in Section 6.

As Chronos is based on the implicit path enumeration technique (IPET) [Li 1995], the output of Chronos is an integer linear programming (ILP) problem: a set of integer linear equations that represent constraints, and an objective function to be maximised subject to the constraints. The solution to the ILP problem is the worst-case execution time, and is obtained using an off-the-shelf ILP solver.

Additional constraints can be added manually in order to exclude certain infeasible paths. As the analysis only considers the control flow graph, it has no insight into the values of variables in the kernel. Therefore, it may consider paths that are not actually realisable. In order to exclude these paths, we manually added extra constraints to the ILP problem where necessary. These constraints take one of three forms:

- a **conflicts with** b **in** f: specifies that the basic blocks at addresses a and b are mutually exclusive, and will not both execute during an invocation of the function f. If f is invoked multiple times, a and b can each be executed under different invocations.
- a **is consistent with** b **in** f: specifies that the basic blocks at addresses a and b will execute the same number of times during an invocation of the function f.
- a **executes** n **times**: specifies that the basic block at address a will only ever execute at most n times in total in all possible contexts.

It would be possible to transform these extra constraints into *proof obligations* – statements which a verification engineer could be asked to prove formally. This would remove the possibility of human error mistakenly excluding a path which is in fact feasible, resulting in an unsound analysis.

5.3 Computing Loop Bounds

As described in the previous section, our analysis currently requires annotations to specify the iteration counts of each loop in seL4. Many of these annotations can be generated automatically. For example, loops which use explicit counter variables can be easily bounded using static analysis. We have computed the loop bounds automatically for several of the loops in seL4 using program slicing and model checking, thereby reducing the possibility of human error. Our approach shares similarities to previous model-checking techniques to find loop bounds [Rieder 2008] or compute WCET [Metzner 2004]. It operates on unmodified binaries, and is unaffected by compiler optimisations.

First, we obtain the semantics of each individual program instruction using an existing formalisation of the ARMv7 instruction set [Fox 2010]. We transform the program into SSA form [Cytron 1991], and then compute a program slice [Weiser 1984] to identify a subset of instructions which encapsulates the behaviour of the loop. This slice captures the control flow dependencies of the loop, and expresses the semantics of the data flow through the loop. This information is converted into a model in linear temporal logic (LTL). A model checker is used to solve for the maximum execution count of the loop head by using a binary search over the loop count.

Not all loops in seL4 have been analysed successfully yet – due to the lack of pointer analysis support in our tools, we presently are unable to compute the bounds of loops which store and load critical values to and from memory. However, almost all of these accesses are to and from the stack, which in the absence of pointer aliasing (guaranteed by seL4's invariants), can be computed and tracked offline. With some work, we expect that this technique will be able to compute all of the remaining loop bounds in seL4.

There are other types of loops which are difficult to bound – e.g., loops which traverse linked lists. In seL4, linked lists do exist which are bounded only by the size of physical memory. However, all traversals of these lists contain preemption points after a fixed number of iterations. Therefore, for interrupt response time analysis, we can simply consider the fixed number of iterations as the upper bound.

5.4 Comparing Analysis with Measurements

The results of our static analysis give an upper bound on the worst-case execution time of kernel operations. However, this upper bound is a pessimistic estimate. In practice, we may never observe latencies near the upper bound for several reasons:

- The conservative nature of our pipeline and cache models means we consider potential worst-case processor states which are impossible to achieve.
- Of the worst-case processor states which are feasible, it may be extremely difficult to manipulate the processor into such a state.
- The worst-case paths found by the analysis may be extremely difficult (but not impossible) to exercise in practice.

In order to gain some insight into how close our upper bounds are, we can try to approximate the worst-case as best we can. We wrote test programs to exercise the longest paths we could find ourselves (guided by the results of the analysis) and ran these on the hardware. Our test programs pollute both the instruction and data caches with dirty cache lines prior to exercising the paths, in order to maximise execution time. We measured the execution time of these paths using the cycle counters available on the ARM1136's performance monitoring unit.

In order to quantify the amount of pessimism introduced by the first two factors alone, we used our static analysis model to compute the execution time of paths that we are able to reproduce, and compared them with real hardware. The results of this are shown in Section 6.2.

Event handler	Before changes; L2 disabled	After changes; L2 disabled			After changes; L2 enabled		
	Computed	Computed	Observed	Ratio	Computed	Observed	Ratio
System call	3851 μs	332.4 μs	101.9 μs	3.26	436.3 μs	80.5 μs	5.42
Undefined instruction	394.5 μs	44.4 μs	42.6 μs	1.04	76.8 μs	43.1 μs	1.78
Page fault	396.1 μs	44.9 μs	42.9 μs	1.05	77.5 μs	41.1 μs	1.89
Interrupt	143.1 μs	23.2 μs	17.7 μs	1.31	44.8 μs	14.3 μs	3.13

Table 2. WCET for each kernel entry-point in seL4, before and after our changes to reduce WCET. Computed results are a safe upper bound on execution time. Observed results are our best-effort attempt at recreating worst-cases on hardware.

6. Results

We first computed the worst-case execution time of our modified seL4 kernel binary using only the loop iteration counts and no other human input. This provided us with an upper bound on the execution time of the kernel of over 600,000 cycles. We converted the solution to a concrete execution trace. However, from looking at these traces it was quickly apparent that the solution was in fact infeasible as the path it took was meaningless – no input could possibly result in execution of the path.

We then added additional constraints of the form described in Section 5.2, in order to eliminate paths that were obviously infeasible. Each of these constraints was derived by observing why a given trace could not be executed. The biggest cause of these infeasible paths was due to the style of coding in seL4, which stems from its functional roots. Many functions in seL4 contain switch statements that select code based on the type of cap passed in, as shown in Figure 6. If `f()` and `g()` both use this style then, due to virtual inlining, much of the inlined copy of `g()` will never be executed, as the return value of `getCapType()` is guaranteed to be the same in both functions.

Our analysis detects the WCET of `g()`, which only occurs for one specific cap type, as contributing to every invocation of `g()` from `f()`. This leads to significant overestimation of the WCET. Based on this, we added several constraints of the form *a is consistent with b*, where a and b were the blocks corresponding to the same types in `f()` and `g()`.

We added additional constraints until we obtained a path that appeared to be feasible. This path has an execution time estimate of 232,098 cycles with the L2 cache enabled, or 176,851 cycles with the L2 cache disabled. On the 532 MHz i.MX31, this corresponds to an execution time of 436.3 μs with L2 and 332.4 μs without L2.

The full results of the analysis are shown in Table 2. The first column shows the WCET before we modified seL4 as outlined in this paper, making the results comparable to our previous analysis [Blackham 2011a] (the previous analysis was on the OMAP3 platform and thus is not directly comparable). For the system call path, a factor of 11.6 improvement in WCET was observed, largely due to the added preemption points. The other kernel entry points also see a significant improvement because the scheduler bitmaps and the new address-space management techniques remove two potentially long-running loops.

```
void f(cap_t cap) {
  ...
  switch (getCapType(cap)) {
      case frame_cap:
          ...
          g(cap);
          ...
          break;
      case page_table_cap:
          ...
          g(cap);
          ...
          break;
      ...
```

Figure 6. Example of the coding style used in many seL4 functions. This arises from the use of pattern matching over type constructors in its Haskell specification.

The worst-case interrupt latency of seL4 is the sum of the WCET for the system call path (the longest of all kernel operations), and the interrupt path. This gives an upper bound on the interrupt latency of 481 μs with L2 and 356 μs without.

For all entry points except the system call handler, we were able to construct scenarios that produced execution times that were within 31% of the computed upper bound when the L2 cache was disabled. Enabling the L2 cache increases the amount of pessimism in our upper bounds, and thus the disparity is much higher (e.g. 3.13 for the interrupt path). Recreating the path identified by the system call handler proved to be extremely complicated and our best efforts only achieved a case that was within 5.4 times of our computed upper bound.

6.1 Analysis

The worst-case we detected was a system call performing an atomic two-way (send-receive) IPC operation, using all the features seL4 provides for its IPC, including a full-length message transfer, and granting access rights to objects over IPC. The largest contributing factor to the run-time of this case was address decoding for caps. Recall that caps are

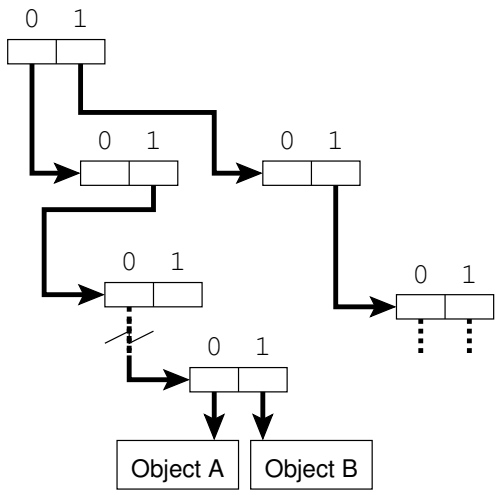

Figure 7. A worst-case address decoding scenario uses a capability space that requires a separate lookup for each bit of an address. Here, a binary address `010...0` would decode to object A, but may need to traverse up to 32 levels of this structure to do so.

essentially pointers to kernel objects with some associated metadata. In seL4, caps have addresses that exist in a 32-bit *capability space*; decoding an address to a kernel object may require traversing up to 32 edges of a directed graph, as shown in Figure 7. In a worst-case IPC, this decoding may need to be performed up to 11 times, each in different capability spaces, leading to a huge number of cache misses. Note that most seL4-based systems would be designed to require at most one or two levels of decoding; it would be highly unusual to encounter anything close to this worst-case capability space on a real system, unless crafted by an adversary.

This worst case demonstrates that our work has been successful in minimising the interrupt latency of the longer running kernel operations, such as object creation and deletion. In previous analyses of seL4, a distinction was made between *open* and *closed* systems, where closed systems permitted only specific IPC operations to avoid long interrupt latencies, and open systems permitted any untrusted code to execute. Our work now eliminates the need for this distinction, as the latencies for the open-system scenarios are no more than that of the closed system.

The atomic send-receive operation exists in seL4 primarily as an optimisation to avoid entering the kernel twice for this common scenario – user-level servers in an event loop will frequently respond to one request and then wait to receive the next. If necessary, the execution time of this operation could be almost halved either by inserting a preemption point between the send and receive phases, or by simply forcing the user to invoke the send and receive phases separately. This latter approach would be detrimental to average-case throughput. It would be worthwhile to investigate inserting a preemption point here to reduce worst-case latency even further.

Other entry points to the kernel show no unexpected pathological cases; these entry points are largely deterministic and have little branching. Like the IPC operations, the worst case for these require decoding a capability that exists 32 levels deep in the capability space. However only one such capability needs to be decoded in the other exception handlers (to the thread which will handle the exception).

The improvements outlined in this paper do not significantly affect the best- or average-case execution time. This is because IPCs are the most frequent operations in microkernel-based systems. seL4 already provides *fastpaths* to improve the performance of common IPC operations by an order of magnitude – fastpaths are highly-optimised code paths designed to execute a specific operation as quickly as possible. The fastpath performance is not affected by our preemption points. In fact, the IPC fastpath is one of the fastest operations the kernel performs (around 200-250 cycles on the ARM1136) and hence there would be no benefit to making it preemptible.

6.2 Conservatism of Hardware Model

Our hardware model is conservative to guarantee a safe bound on execution time. In order to determine the amount of pessimism this adds to our upper bounds, we computed the execution time of the specific paths we tested in our analysis. We achieved this by adding extra constraints to the ILP problem to force analysis of the desired path. The results are shown in Figure 8. The observed execution times were obtained by taking the maximum of 100,000 executions of each path.

The disparity between the computed and observed time is attributable to both conservatism in our pipeline and cache models of the processor, and the difficultly in forcing a worst-case processor state. Our model of the CPU's caches is very conservative – the hardware's L1 caches are 4-way set-associative and the L2 cache is 8-way set-associative, but because we have not modelled the replacement policy of the caches, we are forced to treat any contention for cache lines within a cache set as a miss. Thus accesses which might not miss the cache on hardware must be assumed to miss in our model unless it was the most recently-used cache line within the cache set.

As the system call path is much longer than the other three paths (by an order of magnitude), there is much more contention within each cache set. Therefore it suffers the most from this conservative model.

6.3 Computation Time

The entire static analysis ran in 65 minutes on an AMD Opteron (Barcelona) system running at 2.1 GHz. We repeated all analysis steps for each entry point – system calls, undefined instructions, page faults and interrupts. The analy-

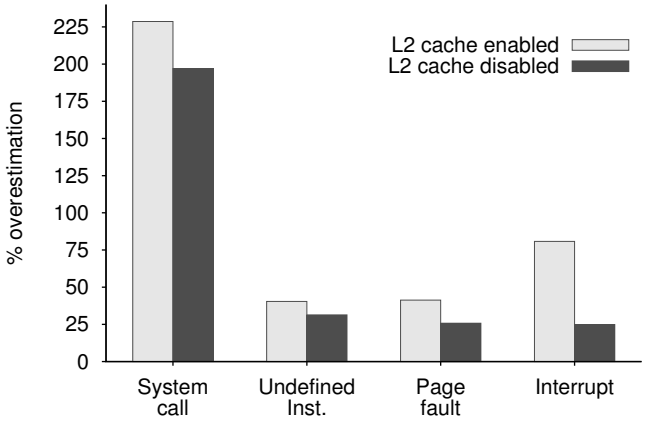

Figure 8. Overestimation of our hardware model for static analysis with the L2 cache enabled and disabled. Each bar corresponds to a realisable path and shows the percentage difference between the observed execution time on hardware and the predicted execution time for the same path.

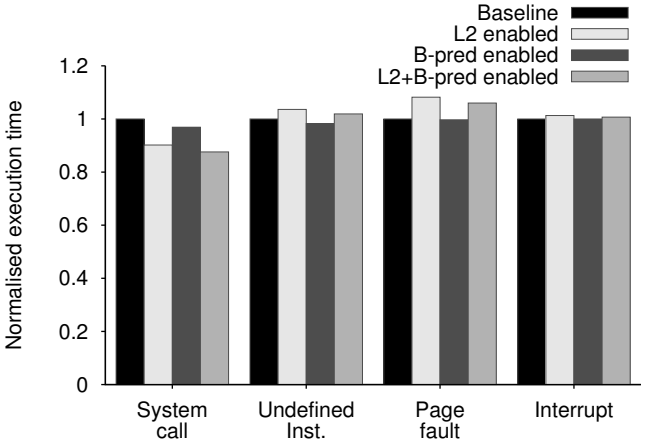

Figure 9. Effects of enabling L2 cache and/or branch prediction on worst-case observed execution times. Each path is normalised to the baseline execution time (L2 and branch predictors disabled).

sis of the latter three entry points completed within seconds, whilst the analysis of the system call entry point took significantly longer. This is to be expected, as the system call handler is the entry point for the majority of seL4's code.

For the system call handler, the most computationally-intensive step of the analysis was running Chronos, taking 61 minutes. Over half the execution time of Chronos was spent in the address and cache analysis phases – these phases compute worst-case cache hit/miss scenarios for each data load, store and instruction fetch.

We went through numerous iterations of adding additional constraints to the ILP problem in order to exclude infeasible paths, with each iteration taking around an hour to execute. Future work will investigate how to automate the generation of these additional constraints to arrive at a feasible solution sooner.

6.4 Impact of L2 and Branch Predictors

As mentioned in Section 5.1, we disabled the branch predictors on our platform and in our model as we are presently unable to analyse their effect. We also compare the effects of enabling or disabling the L2 cache. Figure 9 shows the impact of enabling these features both individually and together on actual execution time.

It is interesting to note that some of the observed times actually increased when enabling the L2 cache, by up to 8% for the page fault path. This is because the worst case scenarios execute with cold caches that are polluted with data which must first be evicted. Enabling the L2 cache increases the latency of the memory hierarchy – from 60 cycles to 96 cycles for a miss serviced by main memory – which is particularly detrimental to cold-cache performance. The code paths executed in seL4 are typically very short and non-repetitive, thereby gaining little added benefit from the L2 cache that is not already provided by the L1 caches.

Enabling the branch predictor gave a minor improvement in all test cases. The benefit is minimal again because of the cold-cache nature of these benchmarks; the benefit of the branch predictor barely makes up for the added costs of the initial mispredictions.

Despite these results, the L2 cache and branch predictors greatly improve performance in the average case. Reduced run-time translates directly to increased battery life for hand-held devices such as mobile phones, and so the slightly detrimental effect on interrupt latency is almost certainly worthwhile on such devices.

As noted earlier, it would be possible to lock the entire seL4 microkernel into the L2 cache. This would result in a huge benefit to worst-case interrupt latency of the kernel whilst also reducing non-determinism, resulting in a tighter upper bound.

7. Related work

Ford et al. explore the differences between process-based and event-based kernels, and present the Fluke kernel, which utilises restartable system calls rather than continuations to implement an atomic kernel API [Ford 1999]. They outline the advantages of this model, including ease of userspace checkpointing, process migration and aided reliability. They also measured the overhead of restarting kernel operations to be at most 8% of the cost of the operations themselves.

Several subprojects of Verisoft have attempted to formally verify all or part of different OS kernels. They have verified a very small and simple time-triggered RTOS called OLOS [Daum 2009b], as well as parts of their VAMOS microkernel [Daum 2009a]. These kernels are also based on the event-driven single-kernel-stack model. They are much

simpler kernels than seL4 (e.g. VAMOS supports only single-level page tables) and are designed for the formally-verified VAMP processor [Beyer 2006]. Whilst the VAMP processor is a real-world product, it is not widely used.

A related project began to verify the PikeOS RTOS with some progress but the proof has not been completed [Baumann 2010]. PikeOS is a commercial product used in safety-critical real-time systems such as aircraft and medical devices, but there has been no indication of a sound worst-case interrupt latency analysis.

Some progress has been made towards verifying code that executes in the presence of interrupts. Feng et al. have constructed a framework on which to reason about OS code in the presence of interrupts and preemption [Feng 2008]. Gotsman and Yang have also constructed frameworks for verifying preemptible and multiprocessor kernels, and have used theirs to verify an OS scheduler [Gotsman 2011].

We refer the reader to Klein's thorough overview of the state of formal verification of operating systems [Klein 2009a].

Our previous analyses of seL4's response time were performed on older versions of the seL4 kernel and targeted an 800 MHz OMAP3 CPU [Blackham 2011a;b]. In this work, we chose a different platform, the i.MX31, in order to benefit from better cache management. The OMAP3 differs from the i.MX31 in CPU speed, micro-architecture and memory latency. We have repeated our previous analysis on the i.MX31 platform to obtain directly comparable results. The work presented here gives a further factor of 11.6 improvement over our previous analysis when the L2 cache is disabled, and explores the verification implications of adding preemption points to reduce interrupt latency.

8. Conclusions and Future Work

We have explored how to reduce the worst-case interrupt response time of a verified protected microkernel such as seL4. We have added preemption points into some of seL4's operations, and have restructured others in order to remove all non-preemptible long-running operations from the kernel. These improvements have been guided by an analysis of the kernel's worst-case execution time.

From its inception, the design of seL4 was intended to limit interrupt latency to short bounded operations, although this was not true of the original implementation. Using static analysis to compute interrupt latencies, we could systematically validate and improve the design where necessary.

As a verified microkernel, seL4 imposes additional constraints on how preemption may be introduced. We must take care when adding preemption points to ensure that the effort of re-verifying the kernel is minimised. This effort can be reduced by avoiding unnecessary violations of global kernel invariants, and searching for intermediate states that are generally consistent with the kernel's normal execution.

With careful placement of preemption points, we have eliminated large interrupt latencies caused by longer running operations inside seL4. This enables seL4 to host execution of untrusted code, confined by the capabilities given to it, and still safely meet real-time deadlines within 189,117 cycles. On modern embedded hardware this translates to a few hundred microseconds, which is ample for many real-time applications.

There is scope to reduce this even further, which will be the subject of future work. We will also focus on automating the analysis to reduce the level of human intervention required. Of course, we will also have the work ahead of us to verify our improvements to seL4.

We believe an event-based kernel such as seL4 could realistically attain a worst-case interrupt latency of 50,000 cycles (or approximately 100 μs on a 500 MHz CPU). Due to its small code size, L2 cache pinning can be very effective at reducing latency for instruction cache misses. Another potential improvement is to remove, or make preemptible, seL4's atomic send-receive IPC operation.

The biggest issue with seL4's design in terms of interrupt latency, is the decision to use a versatile 32-bit capability addressing scheme – each of the 32 bits that need to be decoded can theoretically lead to another cache miss, and decoding several such addresses results in significant interrupt latencies. Practical systems can use the seL4 authority model to prevent this scenario by not allowing an adversary the ability to construct their own capability space.

We have shown that the event-based model of seL4 does not lead to unreasonably large interrupt latencies. It also brings the benefits of reduced kernel complexity. We assert that a process-based kernel without preemption would attain interrupt latencies in the same order of magnitude as the event-based model. A fully-preemptible process-based kernel might attain much smaller latencies, but forgoes the possibility of formal verification.

Verification technology may some day be able to reason about the correctness of fully-preemptible kernels. Until then, a formally-verifiable event-based microkernel such as seL4 provides a very high assurance platform on which safety-critical and real-time systems can be confidently built.

Acknowledgements

Many members of the seL4 team helped to develop the ideas presented in this paper, including Kevin Elphinstone (lead architect), Adrian Danis, Dhammika Elkaduwe, Ben Leslie, Thomas Sewell and Gerwin Klein.

We thank our shepherd, Wolfgang Schröder-Preikschat, for his guidance and valuable feedback, and our anonymous reviewers for their helpful comments.

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

[ARM 2005] *ARM1136JF-S and ARM1136J-S Technical Reference Manual*. ARM Ltd., R1P1 edition, 2005.

[Baker 1991] T. P. Baker. Stack-based scheduling for realtime processes. *J. Real–Time Syst.*, 3(1):67–99, 1991.

[Baumann 2010] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Ingredients of operating system correctness. In *Emb. World Conf.*, Nuremberg, Germany, Mar 2010.

[Beyer 2006] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together—formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):411–430, 2006.

[Blackham 2011a] 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.

[Blackham 2011b] Bernard Blackham, Yao Shi, and Gernot Heiser. Protected hard real-time: The next frontier. In *2nd APSys*, pages 1:1–1:5, Shanghai, China, Jul 2011.

[Campoy 2001] M. Campoy, A.P. Ivars, and J.V.B. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop*, 2001. Satellite of the IEEE Real-Time Systems Symposium.

[Cytron 1991] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Progr. Lang. & Syst.*, 13:451–490, October 1991.

[Daum 2009a] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *JAR: Special Issue Operat. Syst. Verification*, 42(2–4):349–388, 2009.

[Daum 2009b] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation correctness of a real-time operating system. In *IEEE Int. Conf. Softw. Engin. & Formal Methods*, pages 23–32, Hanoi, Vietnam, 2009. IEEE Comp. Soc.

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

[Draves 1991] R.P. Draves, Brian N. Bershad, R.F. Rashid, and R.W. Dean. Using continuations to implement thread management and communication in operating systems. In *13th SOSP*, Asilomar, CA, USA, Oct 1991.

[Elkaduwe 2007] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *1st MIKES*, pages 28–34, Sydney, Australia, Jan 2007. NICTA.

[Feng 2008] Xingu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182, Tucson, AZ, USA, Jun 2008.

[Ford 1999] Brian Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *3rd OSDI*, pages 101–115, New Orleans, LA, USA, Feb 1999. USENIX.

[Fox 2010] Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, Jul 2010. Springer-Verlag.

[Gotsman 2011] Alexey Gotsman and Hongseok Yang. Modular verification of preemptive OS kernels. *16th ICFP*, pages 404–417, 2011.

[Heiser 2010] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *1st APSys*, pages 19–24, New Delhi, India, Aug 2010.

[Hohmuth 2002] Michael Hohmuth. The Fiasco kernel: System architecture, 2002. Technical Report TUD-FI02-06-Juli-2002.

[Klein 2009a] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

[Klein 2009b] 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.

[Li 2007] 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.

[Li 1995] Yau-Tsun Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *16th RTSS*, pages 298–307, 1995.

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

[Liedtke 1995] Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[Mehnert 2002] 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.

[Metzner 2004] Alexander Metzner. Why model checking can improve WCET analysis. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification*, volume 3114 of *LNCS*, pages 298–301. Springer-Verlag, 2004.

[Puaut 2002] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *23rd RTSS*, pages 114–123, 2002.

[Rieder 2008] B. Rieder, P. Puschner, and I. Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In *Intelligent Solutions in Embedded Systems, 2008 International Workshop on*, pages 1–7, july 2008.

[Warton 2005] Matthew Warton. Single kernel stack L4. BE thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Nov 2005.

[Weiser 1984] Mark Weiser. Program slicing. *IEEE Trans. Softw. Engin.*, SE-10(4):352–357, Jul 1984.