Towards a User-Mode Approach to Partitioned Scheduling in the seL4 Microkernel

Mikael Åsberg and Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23,
Västerås, Sweden
{mikael.asberg,thomas.nolte}@mdh.se

***Abstract*—This paper presents a preliminary study of applying partitioned scheduling in the seL4 microkernel. This microkernel is the first operating system kernel ever to be formally proven for its functional correctness. Even though the kernel is completely verified it still delivers high performance comparable to other L4 kernels. The seL4 kernel implements *isolation* of components in terms of the memory resource and security. However, there is still a missing part when it comes to isolation and that is time partitioning. Time partitioning can be implemented inside the kernel (privileged mode) or in user space (user mode). The latter is done using regular user-space thread(s) and can easily be modified while the other approach requires re-verification of the kernel whenever modifications to the time-partitioning policy is done. On the other hand, having the time-partitioning mechanism in privileged mode would yield better performance. We have implemented time partitioning (partitioned scheduling) in the seL4 user space and we elaborate on its performance in terms of overhead costs.**

***Index Terms*—hard real-time systems, partitioned scheduling, implementation**

I. Introduction

Introduction Software defects (bugs) is something that is difficult to avoid, especially when the code base is large and complex. Take for example a relative small code base like the (secure embedded L4) seL4 kernel which comprises around 9000 Source Lines Of Code (SLOC). The code base is relatively small compared to, for example, the Linux kernel (2.6.35) that has 13.5 million SLOC [1]. Still, the verification process of the seL4 kernel took 20 person years to perform and it revealed 144 software defects [2]. A system is definitely not safe and reliable if the underlying software platform is not verified. Critical software applications are not reliable if other applications on the same platform can disrupt them through shared resources such as memory, CPU etc. On the other hand, software development in domains such as the automotive industry [3] and the avionics industry [4] strive towards having integrated applications on the same platform.

The key idea with seL4 is simple. First of all, make sure that the code base, which has 100% control of the system, is 100% free from defects. It is not enough to ensure that the application code is correct or if the applications are compositionally developed with well defined components etc. If the kernel crashes then everything crashes. Secondly, make sure that the applications are 100% protected from each other by the means of partitions. Partitoned software is more robust than flat software since defects will only bring down a delimited part of the software and not the whole system. Partitioning is a powerful mechanism and it has been adopted by the avionics industry in form of the ARINC653 [4] software specification. However, isolation is of course difficult to achieve because we are dealing with many sources, e.g., memory, CPU, security etc. seL4 has come a long way when it comes to partitioning. Applications accessing memory are limited to their assigned address space (similar to ARINC653). Also, all system calls to the kernel and Inter-Process Communication (IPC) is strictly controlled by seL4 through a capability-based access-control model. This gives the user the possibility to configure access rights and thereby isolate software components from each other.

Thread scheduling in seL4 is based on priorities. The policy is based on round-robin scheduling of threads when they have the same priority level. In essence, if threads have different priorities then seL4 resembles the scheduling in VxWorks and `SCHED_FIFO` in Linux, i.e., the highest priority thread will run until it performs a blocking operation and thread preemption occurs when higher priority threads become active (when it stops blocking/suspension). Recall that seL4 uses time slices (round robin) for threads with the same priority, hence, this resembles thread scheduling in ARINC653 but limited to one thread per partition and a fixed time slice.

Goal We want achieve protection of real-time applications. We focus on the protection of the applications temporal aspects.

Method We want to use partitions to partition applications, hence, it requires partitioning mechanisms in the scheduling policy. We refer to this partitioning mechanism as *partitioned scheduling* (on a uni-core platform). The term *partitioned scheduling* should not be confused with the corresponding term in the context of multi-core scheduling.

Problem The current thread scheduling in seL4 offers poor time partitioning of real-time applications. At best, threads can be scheduled in individual partitions (one thread per partition) in a fair manner (round robin) which is not a suitable policy for real-time embedded systems. This is illustrated in Figure 1 where task **A** has the highest priority, task **B** and task **C** have

the same priority level, and task **D** has the lowest priority. Tasks **B** and **C** are scheduled in a fair manner using round robin between time **z** and **x** (since they have the same priority). We see that tasks **B** and **C** are scheduled using enforcement (which resembles partitions) in order to implement round-robin time-slices. If threads have different priority levels then there is no support for partitioned scheduling. This is illustrated in Figure 1 between time **w** and **z**, and also between time **x** and **y**.

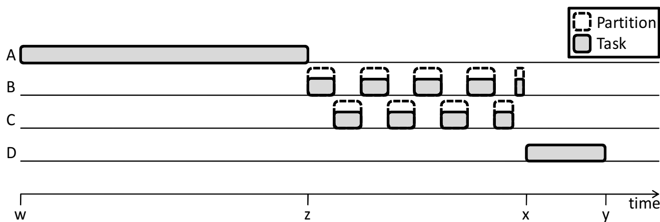

Fig. 1. Scheduling in seL4.

We aim at giving seL4 a flexible scheduling policy (well suited for real-time systems) by supporting time partitioning of applications. This form of scheduling is illustrated in Figure 2 where all tasks are confined inside a partition (constrained by the partition budget) and the partitions are scheduled periodically according to Earliest Deadline First (EDF). Partitioned scheduling is the only lacking piece in seL4 which would make it a complete resource-partitioning aware kernel.

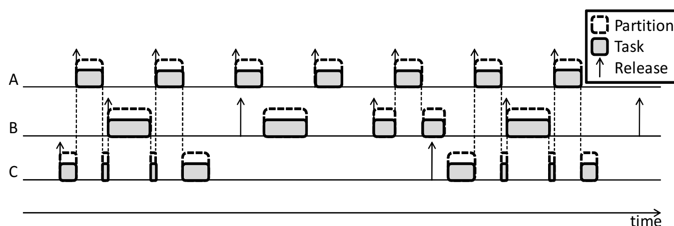

Fig. 2. Partitioned scheduling on uni-core processor.

Possible solutions There are in essence 2 ways of supporting partitioned scheduling in seL4:

1) Implement the partitioned scheduler (PS) in user mode. *Poor performance but flexible solution.*
2) Implement the PS in privileged mode. *Good performance but static solution.*

We see that there is a performance difference between (1) and (2). This is obvious since solution (1) implements the scheduler in a user thread while (2) can keep the scheduler inside the kernel itself, hence, less thread context-switching. The downside with solution (2) is that any change of the scheduling policy implies re-verifying the seL4 kernel. Hence, we are more or less stuck with one or a few defined policies.

In this paper we adopt to solution (1). The obvious reason behind this choice is that we cannot access the seL4 kernel source code. But even if we could, we do not possess the knowledge of re-verifying a kernel with 8700 SLOC using the Isabelle/HOL theorem prover. Hence, we do not have a choice. However, as future work, it would be very interesting to measure how much the performance difference is between these two solutions (1 and 2). This is the main driver behind this paper.

Our main goal is to develop a verified partitioned-scheduler for the seL4 user space. One important feature is that the scheduler should have good performance (otherwise we ruin the whole idea with seL4 since it is a high performance microkernel). We have previously developed [5] such a verified partitioned scheduler for VxWorks. However, it has poor performance and a large model (with many states and transitions) since it is based on timed automata. We intend to develop a new scheduler but based on a different language than timed automata. The first obstacle is to find out if it is possible to implement partitioned scheduling in the seL4 user space. Secondly, if possible in user space, implement a (manually coded) prototype scheduler and observe the overhead.

Contribution In this paper we present a prototype PS implemented for the seL4 microkernel. We will present exact overhead measurements (with CPU cycle accuracy) of the implemented scheduler itself and other related system overheads. We cannot compare this solution to (2) since such a scheduler does not exist yet but this prototype scheduler can be used later as a reference when comparing against a verified (user space) version. We consider the implementation presented in this paper as our baseline.

Outline The outline of this paper is as follows: In Section II we present the preliminaries. Section III presents the related work. Further, Section IV describes the scheduler implementation and, in Section V, we evaluate the overhead of our solution and related system overheads. Finally, Section VI concludes our work.

II. PRELIMINARIES

Partitioning is, as mentioned, the primary method that seL4 uses to tackle software complexity since partitioning facilitates verification and verification itself facilitates the handling of software complexity. The partitioning mechanism is also used by the avionics industry (ARINC653) to build safe systems. When it comes to partitioning of time, i.e., the CPU resource, a common framework used (for example in ARINC653) is a two level (hierarchical) scheduling scheme as depicted in Figure 3. We can observe that there are two kinds of schedulers; global and local. The global scheduler is responsible for scheduling partitions and the local scheduler (if any) handles scheduling of threads (tasks). The scheduling policy at any level can be arbitrary. For example, ARINC653 defines static time-table scheduling at the global level and Fixed-Priority Preemptive Scheduling (FPPS) of periodic threads at the local level.

The main advantage with hierarchical scheduling is the run-time mechanism that divides the CPU cycles among groups of threads (a partition) instead of giving CPU resource distribution at the level of threads which is common in most operating systems. Hence, the CPU resource is distributed at the level of applications which is more suitable if we let different applications share the same CPU. Examples of

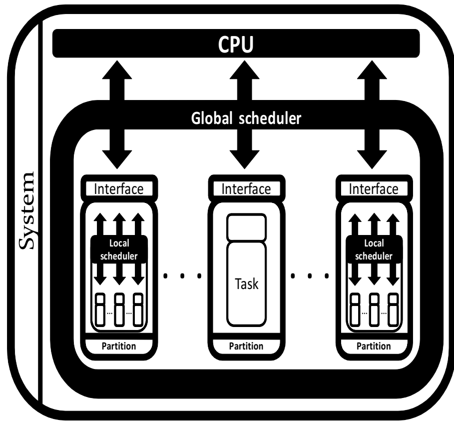

Fig. 3. Hierarchical Scheduling Framework.

applications can be a Virtual Machine (VM) or an engine control management system. The division of the CPU becomes easier, as well as the analysis, and we get a form of fault isolation within partition bounds. Another benefit is that well defined partitions with clear interfaces have the advantage that they are easier to reuse in other systems.

III. Related work

A significant amount of work [6], [7], [8], [9], [10], [11], [12], [13] has focused on analyzing hierarchically scheduled systems, which initially originated from the open systems principle [14] back in the 90's. Open-systems analysis such as the work from Shin *et al.* [8] fits well with our implementation.

a) Scheduler implementations: The first papers dealing with resource reserves of the CPU was Wang *et al.* [15] and Oikawa *et al.* [16] (1999). Both approaches are based on modifications of the Linux kernel in order to enhance the real-time capabilities by introducing some form of CPU reserves. Kim *et al.* [17] proposed the SPIRIT-μKernel back in year 2000 that implemented a two-level hierarchical FPPS framework. The next year, Regehr *et al.* [18] presented an implementation of hierarchical scheduling in Windows 2000. In 2005, Lin *et al.* [19] implemented a scheduler (Vsched) that could schedule periodic type-2 virtual machines in Linux without requiring modifications to the kernel. "Hijack" [20] (by Parmer *et al.*, 2007) is a resource reservation module for Linux which does not require any modifications to the kernel itself. SCHED_DEADLINE by Faggioli *et al.* [21], [22] (2009) is a scheduler that implements EDF scheduling of partitions in Linux. AQuoSA (Adaptive Quality of Service Architecture) by Palopoli *et al.* [23] (2009) is a resource reservation scheduler for Linux based on feedback. More recently, Behnam *et al.* [24] (2008), Heuvel *et al.* [25] (2009) and Inam *et al.* [26] (2011) implemented two-level hierarchical FPPS in the commercial real-time operating systems VxWorks, μC/OS-II and FreeRTOS respectively. Yang *et al.* [27] (2011) implemented a two level hierarchical scheduler in the L4/Fiasco microkernel running L4Linux virtual machines on top. Åsberg *et al.* [28] (2012) presented the ExSched scheduling framework which is capable of hierarchical and multi-core scheduling in Linux and VxWorks without requiring kernel modifications. Molnos *et al.* [29] presents an implementation of a light-weight RTOS with two-level scheduling running on top of a SoC platform. The global scheduler resides within the RTOS itself along with a few local schedulers. The RTOS and all of its schedulers are claimed to be verified. Applications (in this case H264 and JPEG decoders) can either use their own (un-trusted) local scheduler or one of the verified local schedulers in the RTOS. The global scheduler is in charge of scheduling the applications.

b) Scheduler modelling/verification: Few papers touch upon the field of modelling and verification of hierarchical (partitioned) scheduling. Muller *et al.* [30], [31] presented Bossa in 2002/2004. This framework is used for scheduler development and has a domain specific language (DSL) which can model schedulers (such as hierarchical schedulers). Bossa supports scheduler synthesis for early Linux kernel versions. Ha *et al.* [32] (2004) presented theorem-proving verification of the Integrated Modular Avionics (IMA) scheduler in the DEOS kernel (which is used in safety-critical domains). This scheduler assigns a period and a time slice to each thread and schedules them using Rate Monotonic (RM). Singhoff *et al.* [33] (2007) presented modelling and schedulability analysis of two-level hierarchical scheduling (using timed automata) in their simulation tool Cheddar. Recently (2011), Åsberg *et al.* [5] presented modelling, verification and synthesis of two-level hierarchical FPPS. The synthesized scheduler was integrated into VxWorks.

IV. Implementation

The PS implementation is based on the seL4 microkernel version 1.1[1]. Compilation was done using the cross-compiler in Sourcery CodeBench Lite 4.6.3.

The PS prototype implements EDF scheduling of partitions where each partition contains one thread each. Hence, this scheduler is identical to the SCHED_DEADLINE [21], [22] scheduler in Linux. We have chosen not to add local scheduling inside the partitions due to technical challenges with implementing the periodic task model [34] in seL4. We defer the work of adding a second layer scheduling to future work.

It is easy to switch the EDF algorithm at the global scheduling level to FPPS instead. Its just a matter of replacing the deadline queue with a thread-priority queue.

Partitions have the parameters period (T), deadline (D) and budget (Θ). The active partition with the smallest absolute deadline will always be the current executing partition, i.e., the priority of the partitions are dynamic depending on the current absolute deadline. The reason for choosing EDF instead of FPPS is because they have the same implementation complexity and almost the same runtime overhead (EDF has one more addition operation than FPPS) but EDF has in general better CPU utilization and it generates less number of preemptions compared to FPPS [35].

[1] seL4 http://www.ertos.nicta.com.au/software/seL4/home.pyl

seL4 has a sealed kernel since it is verified and any modifications would invalidate the verification. Hence, we are forced to implement the scheduler in a user thread. We implemented the EDF PS in the root thread which is the first thread to start at bootup. We set up this thread to be awakened by periodic interrupts (Figure 4).

```
1. while(true) {
2.     // Acknowledge the interrupt.
3.     seL4_IRQHandler_Ack(irq);
4.     // Wait to receive interrupt notifications.
5.     seL4_Wait(endpoint, &sender);
6. }
```

Fig. 4. Main loop in the PS.

We let the root thread be responsible for setting up the partitions and creating threads (to schedule) and connect these to their partitions. The thread body is represented in Figure 5. We save a timestamp in the beginning of its execution so that we can trace and record the execution of threads. Since we do not have access to the kernel we cannot do proper thread-execution recording which is important in order to debug the scheduler.

```
1. void thread(void) {
2.     long long tstamp;
3.     // Timestamp when this thread started.
4.     tstamp = getRDTSC();
5.     log(tstamp);
6.     while (1) ;
7. }
```

Fig. 5. Thread body.

A. Queue management

The core functionality of the EDF scheduler is the release and deadline queue. The release queue keeps the release times ordered with the smallest value first. The ordering will decrease the thread-release overhead, especially since element insertion and retrieval has O(1) complexity in our implementation. The deadline queue has the thread absolute-deadlines stored in a ordered fashion. Hence, the complexity to determine the highest priority thread is also O(1). As mentioned, deadlines are stored as absolute values, the same as for release times. The reason for storing them as absolute values (instead of relative [25]) is because we avoid the complexity of having to decrement the relative time as time progresses. However, the problem with absolute values is that they must wrap around at some point (an unsigned 32 bit integer will for example wrap at 4 294 967 296). We solve this by keeping two data structures of the queue. The values are stored in the second queue whenever there is a wrap around. The queue itself is based on bitmaps, i.e., an element in the queue is represented by a bit in an integer. The queue is illustrated in Figure 6.

As the figure shows, bit 0 in `bitmap[0]` and bit 30 in `bitmap[2]` is set which means that we have the two values 1 and 95 stored in the queue currently. The `bitmap_nodes` structure keeps the information about which threads that are represented in the bitmap queue `bitmap`. For example, there are two threads, both with the value 1. The linked-list structure in `bitmap_nodes` links the thread ID (through the linked nodes) to the bitmap. The nodes are inserted/retrieved in FIFO order in the linked list (this will minimize unnecessary task switches). The `bitmap_nodes` structure could of course be optimized in terms of memory usage by having a `bitmap_nodes` structure with less elements and use a hash function to map between it and the bitmap structure (`bitmap`).

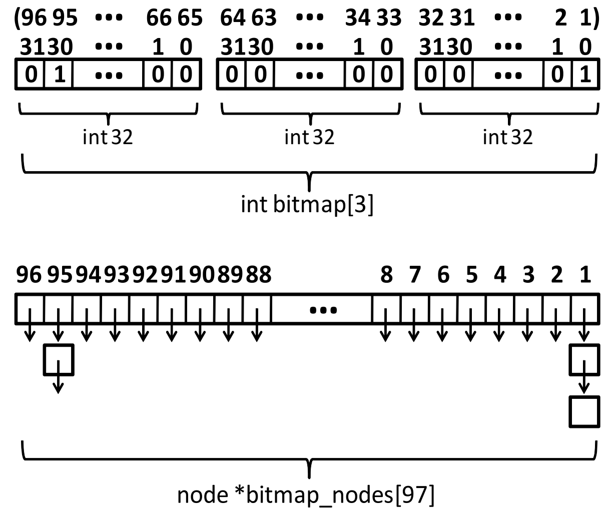

Fig. 6. Representation of the bitmap queue structure.

Recall the discussion about the problem with wrap arounds. To solve this problem we have the data structures in Figure 6 replicated. So for example, assume we have a task with period 95. When the schedule reaches absolute time 95 then we release the task and update its value in the release queue (Figure 6). The next absolute release time for this task will be $95 + 95 = 190$. We wrap the new period value around 96 since the queue in this example can only represent the values $1-96$. Hence, the new period value will be $190-96 = 94$. So we insert the value 94 into the replicated queue. We always keep one queue as the active one, hence, this means that we first retrieve the elements in the active queue. When the active queue is empty then we switch to the second queue and start retrieving elements from this queue instead. Hence, the second queue becomes the active queue. In this manner we form a circular queue that wraps the element values.

The period and deadline values of all the partitions in the system will dictate the length of the bitmap queue. The length of the bitmap queue must be large enough to contain the largest period/deadline value otherwise the wrapping will not work. Using a hash table and/or small period/deadline values may actually incur less memory overhead than non bitmap-based queues. However, if the period/deadline value

distribution is too wide then it could be the other way around. The positive aspect with a bitmap queue is that it only has a time complexity of O(1) for both insertion and retrieval of elements. As a comparison, one of the main Linux kernel queue-structures is the Red Black Tree (RBT) which has a time complexity of O(log n), i.e., worse than bitmap. For example, SCHED_DEADLINE [21], [22] uses RBT.

The reason why the bitmap queue has O(1) time complexity is because it takes a constant time to set a bit in an integer and the same goes for the retrieval of the least significant bit. The time length to perform these two operations are constant independent of the amount of elements (bits) stored in the queue. The algorithm to retrieve the least significant bit is shown in Figure 7. The algorithm systematically detects bits and bit-shifts towards the least significant bit. Using the corresponding CPU instruction `ffs` did not affect our experimental results significantly (Section V). However, using this algorithm instead makes the implementation hardware independent.

```
1. int my_ffs(int the_integer) {
2.
3.     int least_signif_bit = 1;
4.
5.     if ( the_integer == 0 )
6.         return 0;
7.
8.     if ( (the_integer & 0x0000FFFF) == 0 ) {
9.         the_integer >>= 16;
10.        least_signif_bit += 16;
11.    }
12.    if ( (the_integer & 0x000000FF) == 0 ) {
13.        the_integer >>= 8;
14.        least_signif_bit += 8;
15.    }
16.    if ( (the_integer & 0x0000000F) == 0 ) {
17.        the_integer >>= 4;
18.        least_signif_bit += 4;
19.    }
20.    if ( (the_integer & 0x00000003) == 0 ) {
21.        the_integer >>= 2;
22.        least_signif_bit += 2;
23.    }
24.    if ( (the_integer & 0x00000001) == 0 ) {
25.        the_integer >>= 1;
26.        least_signif_bit += 1;
27.    }
28.    return least_signif_bit;
29. }
```

Fig. 7. Algorithm to retrieve the least significant bit in a 32 bit integer.

The intention with this scheduler implementation is to reduce the number of scheduler invocations (hence we have chosen EDF) and to minimize the scheduler execution time (hence the bitmap queues). We did this in order to keep the overhead to a minimum, since having the scheduler implementation in user space is not efficient in general.

V. Evaluation

This section presents our experiments with the PS implementation. We will present the measured overhead of the scheduler itself and seL4 context switches. Further, we also provide execution traces of threads scheduled by the PS and we visualize these using the Grasp tool [36].

A. Hardware and software setup

Our PS implementation was executed in the Quick EMUlator[2] [37] (QEMU), version 0.13.91, running on Linux openSUSE 11.4. QEMU is an open-source machine (processor) emulator that emulates real hardware accurately down to CPU cycle level. We configured QEMU to emulate an Intel 533 MHz Pentium3 Katmai processor (model 7, stepping 3). We chose Pentium3 Katmai since it is reliable to use its timestamp counter for time measurements. We will elaborate more on this in the next section.

B. Time measurement

We chose to use the ReaD Time-Stamp Counter (RDTSC) processor register (only for x86 architecture) for accounting time since we wanted a low overhead (and high resolution) facility for time measurements. The RDTSC instruction returns the processor timestamp from a CPU register. This register records the number of CPU clock-cycles since the processor was last reset. However, there are well known issues which can make RDTSC timestamping unreliable.

1) In multi-core architectures, cores may have different values in their RDTSC registers. Hence, threads that migrate from one CPU to another might read incorrect timestamps since registers on different CPUs are not synchronized.
2) Dynamic frequency scaling (called SpeedStep on x86) will change the elapsed time between clock cycles, hence, the counter value becomes unreliable.
3) Out-of-order execution can change the location of the RDTSC timestamp in the source code. Hence, the timestamp may happen earlier or later than intended, giving incorrect time measurements.

In order to tackle these issues we chose a processor (Pentium3 Katmai) with only one core and no SpeedStep. Before each call to RDTSC we put a CPUID instruction-call to flush the instruction pipeline. This will serialize the instruction queue in order to prevent out-of-order execution of the RDTSC operation.

C. Overhead measurements

Figure 8 shows the measured overhead of the PS (without rollback), PS using rollback when resuming threads (rollback is used for tracing purposes, see Section V-D) and the seL4 context switch when switching from the scheduler thread to another thread. Rollback means that a resume of a thread will re-start the thread from its first instruction in its sequence of code. Without the use of rollback, threads are resumed to the last instruction they were at, prior to the thread preemption. If rollback is not used then we can not perform task tracing (for debugging purposes) since the trace point is located prior to the first task instruction (on all tasks). A context switch using rollback will re-start the task at the trace point, hence, this will enable task tracing and the context switch will be registered (see line 4 and 5 in Figure 5).

[2]QEMU www.qemu.org/

The server (partition) parameters were as follows; periods (T) were set randomly to 11, 12, 14, 17, 19, 21, 23, 25 and 28 time units while the budget (Θ) was set to 1 time unit on all servers. Maximum server utilization (with 9 servers) is approximately 52.5% and the time length that we ran the systems and measured overhead was 5000 time units (seL4 ticks). We chose maximum 9 servers due to a limitation of the memory allocation in seL4.

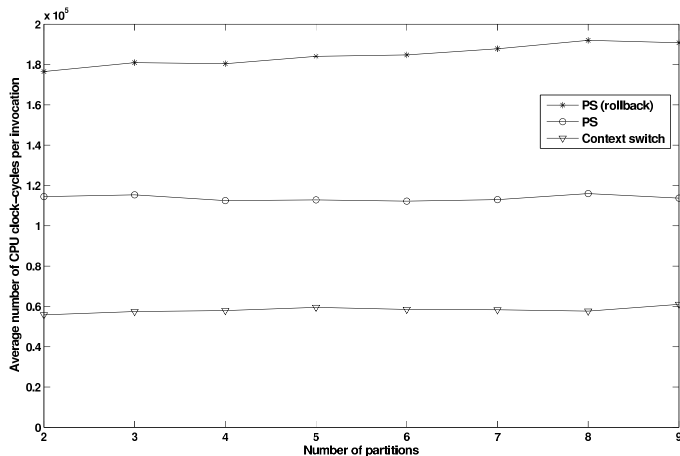

Fig. 8. Average overhead measurements of the PS (with and without rollback) and seL4 thread context-switches.

The context switch was triggered by the `seL4_Wait()` call from the scheduler thread. The figure shows the average amount of CPU cycles it took at each scheduler invocation and context switch for different server (partition) configurations ($2-9$). The context switch overhead should be constant relative to the number of servers running. We see a slight increase which is likely due to measurement uncertainties such as cache effects etc. However, the scheduler overhead of the PS (with and without rollback) should increase as the number of servers increase since more servers increase the risk of multiple scheduling jobs (server releases and server budget-depletions) happening at the same time during a scheduler invocation. Simultaneous scheduling jobs, for example several server releases, during a scheduler invocation will increase the measured average time represented in Figure 8. The difference in scheduler overhead (PS with rollback) when we compare the configuration of 2 and 9 servers is only 14283 clock cycles (26.8 µs) which is relatively small. This is due to the effective O(1) queue management (Section IV-A) in the PS. The overhead of PS without rollback is also increasing with the number of servers but at a lower rate. The average number of clock cycles for a scheduler invocation (with rollback) is 184661 (346 µs). PS without rollback is in average 113747 clock cycles (213 µs) per invocation which may be perceived as long. On the other hand it is only two times the time length of a context switch which is in average 58282 clock cycles (109 µs).

As a comparison, the overhead of setting a timer (at the global scheduling level) in a hierarchical scheduling framework [27] in L4/Fiasco takes in average 236 µs on a 2GHz AMD Athlon processor. Another comparison is the measured time of a system call (`seL4_Send()`, `seL4_Wait()`, `seL4_ReplyWait()`) in seL4 on an ARM Cortex-A8 800MHz which takes approximately 20 µs [38]. Another example from [38] is the time it takes from the arrival of an interrupt until a thread handler starts. This time was measured to be 59.5 µs and 318.3 µs depending on whether certain system calls were allowed (open system) or not (closed system). [39] reported on inter- and intra-process communication (IPC) overheads of 54 µs and 35 µs respectively, running on an ARM1176 (416MHz) processor (with L4/Fiasco). The comparisons are summarized in Table I.
Conclusively, it is difficult to draw any final conclusions from our measurements. The comparisons we have made relate to general system overheads in the seL4 and L4/Fiasco kernels. Based on this, the overhead of the PS (without rollback) does not seem overwhelming, i.e., this overhead is at least not orders of magnitude larger than general system overheads in seL4/L4/Fiasco kernels.

Measurement	Platform	Time (µs)
PS (with rollback)	Intel Pentium3 533MHz (seL4)	346
PS	Intel Pentium3 533MHz (seL4)	213
Context switch	Intel Pentium3 533MHz (seL4)	109
Set timer in HSF [27]	AMD Athlon 2GHz (L4/Fiasco)	236
System call [38]	ARM Cortex-A8 800MHz (seL4)	20
Interrupt delivery [38]	ARM Cortex-A8 800MHz (seL4)	59/318
IPC [39]	ARM1176 416MHz (L4/Fiasco)	35/54

TABLE I
OVERHEAD COMPARISON.

Figure 9 shows the execution trace when the PS interrupts an executing thread to perform a scheduling operation. It is then followed by the seL4 context-switch execution. The context switch was measured by timestamping the end of the PS and the start of the next running thread. It is not possible to measure the time (of the context switch execution) between an interrupted thread and the PS due to that the interrupted thread cannot timestamp the time when it gets interrupted.

D. Execution trace

Figure 10 shows a thread execution trace scheduled with our EDF PS. The threads (denoted task1, task2 and task3 in this figure) partitions had the period (T) values 3, 4 and 5. The budget (Θ) values were set to 1, 1 and 2 respectively.

As mentioned in Section V-B, we used the RDTSC instruction to timestamp the start and end of the thread execution. These timestamp points were placed at the beginning of the thread body and at the beginning and end of the PS code. We also created a background thread with the lowest priority. This thread represents the part of the trace when the scheduled threads are not running. In order to be able to trace thread preemptions, we simply restart (rollback) the thread to the beginning of its body such that we can get a timestamp when it gets re-scheduled by the PS.

The trace (Figure 10) was verified by comparing it with a corresponding EDF trace from the TIMES tool [40]. Its

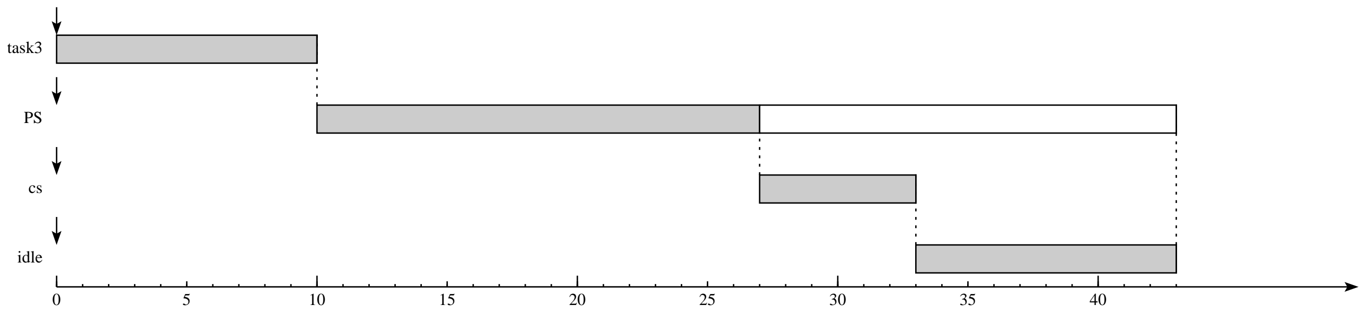

Fig. 9. Execution trace of the PS (with rollback) and a context switch in seL4.

interesting to note that the FIFO order in the thread deadline-queue prevents unnecessary preemptions when deadlines are equal. This can be seen at task3s third instance. task1 is released in the middle of task3s execution with the same deadline (15) but there is no preemption since task3 was released before task1 and hence it is ahead of task1 in the deadline queue at index 15.

VI. Conclusion

The seL4 microkernel is a high performance kernel with the uniqueness that the entire kernel is verified using theorem proving. Hence, this makes the kernel suitable for safety-critical systems. The principle of seL4 is to isolate applications from each other and execute them on a safe and trusted platform. This means that if an application is correctly constructed, in terms of its functionality, then it will never be disrupted or fail due to a faulty kernel or other error-prone applications. One missing piece in the seL4 kernel is the partitioning of time. A solution to this is to put the policy outside of the kernel, i.e., in user space. The downside with this approach is whether it is possible to perform scheduling from user space, and secondly, it will cause a lot of overhead compared to having the policy in the kernel instead. We have aimed at giving answers to these two questions. Can we implement it in user space, and if so, what figures do we get in terms of overhead?

This paper presents a prototype O(1) partitioned scheduler in the seL4 microkernel user-space. The scheduler is based on the earliest deadline first algorithm and it schedules threads periodically (in partitions) delimited by a budget capacity. We have focused on keeping the overhead of the scheduler to a minimum. This is done by choosing the earliest deadline first algorithm to minimize the amount of preemptions, and by selecting time-complexity O(1) queues with FIFO order for equal valued elements.

We have presented the overhead of the scheduler when scheduling $2-9$ partitions. The time length of a scheduler invocation does not increase much when the number of servers is increased. The average overhead of a scheduler invocation is approximately 2 times as large as a context switch (from the scheduler to a thread). The total overhead of a scheduler instance, including a scheduler invocation plus two context switches, is approximately 431 μs. This value can be compared with the observed interrupt latency (from interrupt to thread handler) of 318 μs in an open system running on a platform that is faster than ours (533MHz vs. 800MHz). But still, the overhead is substantial. Two facts can still justify our measured values. First of all, we used an outdated and slow CPU (for the sake of getting reliable measurements). For example, if our platform would have a frequency of 800MHz, then the scheduler-invocation length would be 142 μs. Hence, it would be comparable to the interrupt-latency time. Secondly, we believe that further optimizations could decrease the amount of overhead of our scheduler.

Future work includes optimizing the scheduler for performance gains. If we manage to get it down to a reasonable level then we might consider developing a new scheduler with the same policy, but based on verification using theorem proving. The idea is to use Event-B or Frama-C formal analysis and verification, and try to develop a scheduler that is similar to the scheduler presented in this paper (in terms of its source-code and performance).

References

[1] G. Kroah-Hartman, J. Corbet, and A. McPherson, "Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It," in *The Linux Foundation*, 2009.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proc. of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[3] T. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Mate, and K. Nishikawa, "AUTOSAR - Challenges and Achievements," in *Proc. of the 12th International VDI Congress Electronic Systems for Vehicles*, 2005.

[4] ARINC, *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), 1996.

[5] M. Åsberg, P. Pettersson, and T. Nolte, "Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling," in *Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[6] R. I. Davis and A. Burns, "Hierarchical Fixed Priority Pre-emptive Scheduling," in *Proc. of the 26th IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[7] T.-W. Kuo and C.-H. Li, "A Fixed-Priority-Driven Open Environment for Real-Time Applications," in *Proc. of the 20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[8] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

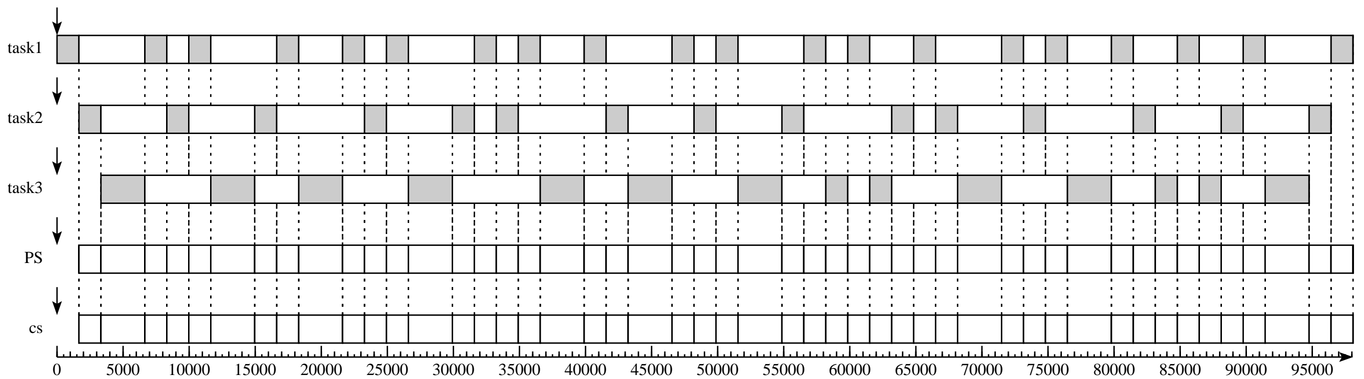

Fig. 10. Execution trace of a set of threads scheduled by the PS (with rollback) in seL4.

[9] X. Feng and A. Mok, "A Model of Hierarchical Real-Time Virtual Resources," in *Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[10] G. Lipari and S. K. Baruah, "Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems," in *Proc. of the 6th IEEE Real Time Technology and Applications Symposium (RTAS)*, 2000.

[11] G. Lipari and E. Bini, "Resource Partitioning Among Real-Time Applications," in *Proc. of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[12] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving Real-time Systems Using Hierarchical Scheduling and Concurrency Analysis," in *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[13] S. Matic and T. A. Henzinger, "Trading End-to-End Latency for Composability," in *Proc. of the 26th IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[14] Z. Deng and J. W.-S. Liu, "Scheduling Real-time Applications in an Open Environment," in *Proc. of the 18th IEEE International Real-Time Systems Symposium (RTSS)*, 1997.

[15] Y. Wang and K. Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *Proc. of the 20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[16] S. Oikawa and R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior," in *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1999.

[17] D. Kim, Y. Lee, and M. Younis, "SPIRIT-uKernel for Strongly Partitioned Real-Time Systems," in *Proc. of the 7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA)*, 2000.

[18] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *Proc. of the 22nd IEEE International Real-Time Systems Symposium (RTSS)*, 2001.

[19] B. Lin and P. A. Dinda, "VSched: Mixing Batch and Interactive Virtual Machines Using Periodic Real-time Scheduling," in *Proc. of the 19th ACM/IEEE International Conference On Supercomputing (SC)*, 2005.

[20] G. Parmer and R. West, "Hijack: Taking Control of COTS Systems for Real-Time User-Level Services," in *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2007.

[21] D. Faggioli, M. Trimarchi, and F. Checconi, "An Implementation of the Earliest Deadline First Algorithm in Linux," in *Proc. of the 24th Annual ACM Symposium on Applied Computing (SAC)*, 2009.

[22] D. Faggioli and F. Checconi, "An EDF Scheduling Class for the Linux Kernel," in *Proc. of the 11th Real-Time Linux Workshop (RTLWS)*, 2009.

[23] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA: Adaptive Quality of Service Architecture," *Software Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[24] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards Hierarchical Scheduling on top of VxWorks," in *Proc. of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2008.

[25] M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual Timers in Hierarchical Real-time Systems," in *Proc. of the WIP session at the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[26] R. Inam, J. Maki-Turja, M. Sjodin, S. Ashjaei, and S. Afshar, "Support for Hierarchical Scheduling in FreeRTOS," in *Proc. of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2011.

[27] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin, "Implementation of Compositional Scheduling Framework on Virtualization," *SIGBED Review*, vol. 8, no. 1, pp. 30–37, 2011.

[28] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," in *Proc. of the 18th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[29] A. Molnos, A. B. Nejad, B. T. Nguyen, S. Cotofana, and K. Goossens, "Decoupled Inter- and Intra- Application Scheduling for Composable and Robust Embedded SoC Platforms," in *Proc. of the 15th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2012.

[30] L. P. Barreto and G. Muller, "Bossa: A Language-Based Approach to the Design of Real-Time Schedulers," in *Proc. of the 10th International Conference on Real-Time Systems (RTS)*, 2002.

[31] J. L. Lawall, G. Muller, and H. Duchesne, "Invited Application Paper: Language Design For Implementing Process Scheduling Hierarchies," in *Proc. of the 11th ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM)*, 2004.

[32] V. Ha, M. Rangarajan, D. Cofer, H. Rues, and B. Dutertre, "Feature-Based Decomposition of Inductive Proofs Applied to Real-Time Avionics Software: An Experience Report," in *Proc. of the 26th International Conference on Software Engineering (ICSE)*, 2004.

[33] F. Singhoff and A. Plantec, "AADL Modeling and Analysis of Hierarchical Schedulers," in *Proc. of the 16th Annual International Conference on the Ada Programming Language (SIGAda)*, 2007.

[34] C. Liu and J. Layland, "Scheduling Algorithms for Multi-Programming in a Hard-Real-Time Environment," *ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[35] G. C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems Journal*, vol. 29, no. 1, pp. 5–26, 2005.

[36] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems," in *Proc. of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.

[37] F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," in *Proc. of the 23rd USENIX Annual Technical Conference (ATEC)*, 2005.

[38] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing Analysis of a Protected Operating System Kernel," in *Proc. of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[39] S. Hessel, F. Bruns, A. Bilgic, A. Lackorzynski, H. Härtig, and J. Hausner, "Acceleration of the L4/Fiasco Microkernel Using Scratchpad Memory," in *Proc. of the 1st Workshop on Virtualization in Mobile Computing (MobiVirt)*, 2008.

[40] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES: A Tool for Modelling and Implementation of Embedded Systems," in *Proc. of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2002.