A Time-composable Operating System*

Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega

University of Padua, Department of Mathematics,
via Trieste, 63 35121 Padua, Italy
`{baldovin,emezzett,tullio.vardanega}@math.unipd.it`

Abstract

Time composability is a guiding principle to the development and certification process of real-time embedded systems. Considerable efforts have been devoted to studying the role of hardware architectures – and their modern accelerating features – in enabling the hierarchical composition of the timing behaviour of software programs considered in isolation. Much less attention has been devoted to the effect of real-time Operating Systems (OS) on time composability at the application level.

In fact, the very presence of the OS contributes to the variability of the execution time of the application directly and indirectly; by way of its own response time jitter and by its effect on the state retained by the processor hardware. We consider *zero disturbance* and *steady behaviour* as those characteristic properties that an operating system should exhibit, so as to be time-composable with the user applications. We assess those properties on the redesign of an ARINC compliant partitioned operating system, for use in avionics applications, and present some experimental results from a preliminary implementation of our approach within the scope of the EU FP7 PROARTIS project.

1998 ACM Subject Classification C.3 [Computer Systems Organization]: Special-purpose and application-based systems – Real-time and embedded systems, D.4.1 [Operating System]: Process Management, D.2.11 [Software Engineering]: Software Architectures – Domain-specific architectures, Patterns

Keywords and phrases Real-time Operating System, Timing composability, ARINC

Digital Object Identifier 10.4230/OASIcs.WCET.2012.69

1 Introduction

The increasing complexity in the design, development and validation of real-time embedded systems can be tackled only is best responded by compositional, incremental software development. The other side of the coin in a compositional approach is that the properties ascertained for individual components in isolation should allow reasoning on the properties of the system that results from their composition (*composability*). Whereas compositionality and composability are consolidated concepts when looking at a system from a purely functional perspective, they are much more difficult to understand and to guarantee when applied to extra-functional concerns and to timing in particular [10]. By the principle of composability, in fact, the timing behaviour of a system should be simply determined as a summation over the execution times of its building blocks; moreover, by composability, a software module should exhibit the same timing behaviour independently of the presence and operation of any other component in the system. Unfortunately, even guaranteeing just timing compositionality on

* This work has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the grant agreement n 249100.

12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012).
Editor: Tullio Vardanega; pp. 69–80
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

current hardware and software architectures is difficult. Although timing analysis frameworks typically characterise the timing behaviour of a system compositionally, a truly composable timing behaviour is not generally provided at the lower levels of a system. The main obstacle to time composability is that modern hardware architectures include a score of advanced acceleration features (e.g., caches, complex pipelines, etc.) that bring an increase in performance at the cost of a highly variable timing behaviour. Since those hardware features typically exploit execution history to speed up average performance, the execution time of a software module is likely to (highly) depend on the state retained by history-dependent hardware, which in turn is affected by other modules. The incurred dependence wrecks composability in the timing dimension as the execution history becomes a characteristic of the whole system and not that of a single component.

For timing compositionality and composability to hold, stringent constraints are imposed on how the system should be conceived and built, in both hardware and software dimensions [9]. Whereas several studies focused on the importance of hardware architectures in enabling compositional timing analysis [6], less attention has been devoted to the role played by other layers in the execution stack. Timing composability is in fact a system property that originates from the underlying hardware and must be preserved across other layers, including the operating system. In this paper we address the role of the operating system layer in preserving timing composability in Integrated Modular Avionics (IMA) systems, where timing composability is a fundamental assumption behind temporal and spatial isolation among software partitions. In particular, we report on our attempt in redesigning part of POK [2], an open-source ARINC653-compliant real-time kernel with a view to timing composability.

The remainder of this paper is organised as follows: in Section 2 we address time composability as a property within the abstraction layers of a typical architecture and discuss a possible approach to enable composability between OS and application layers. Section 3 explains how our approach has been implemented in a partitioned real-time kernel, while Section 4 provides experimental evidence to our arguments. Finally, Section 5 draws some conclusions.

2 Timing composability: a layered approach

Seeking incrementality in the development life-cycle advocates the adoption of an incremental development approach from the system perspective, where incrementality should naturally emerge as a consequence of guaranteeing composability to the elementary constituents (i.e., software modules) of the system. However, in practice, real-time software development can only strive to adopt such discipline, as the supporting methodology and technology are still immature.

We were given the opportunity to study the issues above in the context of the EU FP7 PROARTIS [3] project: this initiative aims at defining a novel, probabilistic framework for timing analysis of critical real-time embedded systems. As an industrial trait, the main project focus is set on partitioned applications, commonly encountered in avionics systems, and particularly on the IMA architecture and its ARINC 653 [1] incarnation. These industrial standards encourage the development of partitioned applications, where the concept of composability is strictly related to the fundamental requirement of guaranteeing spatial and temporal segregation of applications sharing the same computational resources within a federated architecture.

Hardware acceleration features, speculative execution and complex software architectures prevent systems from achieving a composable timing behaviour. Dependence on the execution history, in particular, is one of the main hurdles to timing composability. Removing or at least minimising the effects of history dependence is thus a reasonable approach to achieve a time-composable architecture. Breaking down the execution platform into three classic layers – the hardware layer (`HW PLATFORM`), an operating system layer (`KERNEL`) in the middle, and, finally, a user application layer (`APPLICATION SW`) running on top – makes it possible to address history independence and thus timing composability as a *bottom-up* property that must be first accomplished by the underlying hardware, then preserved across the OS primitives and services, and finally exhibited by the user application.

In this paper, we do not focus on HW-related composability issues, although we are perfectly aware of their relevance, especially with respect to the possible occurrence of timing anomalies [12]. We will assume, instead, the availability of a `HW PLATFORM` where interference from execution history on the timing behaviour has been proactively countered. This is not an unrealistic assumption since it can be achieved by means of simplified hardware platforms [6] or novel probabilistic approaches, as suggested in PROARTIS [3]. Conversely, we consider the `APPLICATION SW` layer, the space within which user applications run, the only level at which (application-logic related) timing variability should be allowed. Ideally, time-composable hardware and kernel layers should be able to remove all history-dependent timing variability so that state-of-the-art timing analysis approach should be able to account for the residual variability at the application level.

The `KERNEL` layer, which is the main focus of our investigation, actually provides an abstraction layer for the operating system primitives and services. From the timing analysability standpoint, the role played by this layer is possibly as important as that played by the `HW PLATFORM`. As a fundamental enabler to compositional analysis approaches, in fact, this layer should preserve the independence property exhibited by the underlying hardware and should not introduce additional sources of timing variability in the execution stack.

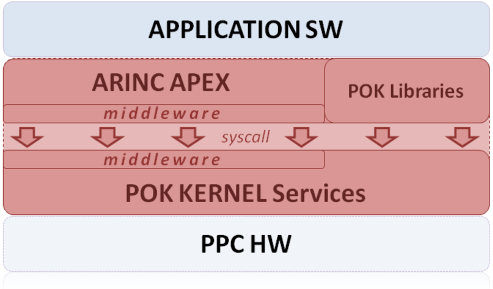


Figure 1 Structural decomposition of POK.

In the scope of our investigation, we focused on the PowerPC processor family, and the PPC 750 model [4] in particular, by reason of its widespread adoption in avionic platforms. We also selected POK [2] as our reference OS kernel because of its lightweight dimensions, its availability in open source and its embryonic implementation of the ARINC specification. We redesigned part of its services with a view to time-composability and analysability. Figure 1 shows a structural breakdown of the POK framework: the `KERNEL` layer provides an interface to a set of standard libraries (e.g., C standard library) and core OS services (e.g., scheduling primitives) to the `APPLICATION SW` layer. In addition, the POK kernel also provides an implementation of a subset of the ARINC Application Executive (APEX).

Enabling and preserving time-composability at the `KERNEL` layer poses two main requirements on the way an OS or ARINC service should be delivered:

- *Zero-disturbance*: in the presence of hardware features that exhibit history-dependent timing behaviour, the execution of an OS service should not have disturbing effects on the application. Some kind of separation is needed to isolate the hardware from the polluting effects of OS or ARINC services. The kind of hardware-level isolation that we seek can be provided by means of techniques similar to those adopted for cache partitioning [8]: a relatively small cache partition should be reserved for the OS so that the execution of OS services would still benefit from the cache acceleration but would not affect the cache state of the user code. However, implementing software cache partitioning (mapping of code to configure separate address spaces) in conjunction with a partitioned OS may result quite cumbersome in practice. An alternative (and easier to implement) approach consists in giving up any performance benefit and simply inhibiting all the history-dependent hardware at once when OS services are executed. This approach, however, comes at the cost of a relevant performance penalty that, though not being the main concern in critical real-time systems, could be still considered unacceptable. Also the execution frequency of a service is relevant with respect to disturbance: services triggered on timer expire (such as, for example, the PowerPC DEC interrupt handler) or an event basis can possibly have even more disturbing effects on the `APPLICATION SW` level, especially with respect to the soundness of timing analysis. The *deferred preemption* mechanism in combination with the selection of predetermined preemption points [13] could offer a reasonable solution for guaranteeing minimal uninterrupted executions while preserving feasibility.

- *Steady timing behaviour*: jittery timing behaviour of an OS service complicates its timing composition with the user-level application. Timing variability at the OS layer depends on a combination of multiple interacting factors: (i) the hardware state, as determined by history sensitive hardware features; (ii) the *software state*, as determined by the contents of its data structures and the algorithms used to access them; and, (iii) the input data. Whereas the first aspect can be treated similarly and contextually with the specular phenomenon of disturbance, the software state instead is actually determined by more or less complex data structures accessed by OS and ARINC services and by the algorithms implemented to access and manipulate them. The latter should thus be re-engineered to exhibit a constant-time – O(1) – and steady timing behaviour, like, for example, constant-time scheduling primitives (e.g., $O(1)$ Linux scheduler [7]). Besides the software state, the timing behaviour of an OS service may be influenced by the input parameters to the service call (so-called input data dependency). This is the case, for example, of ARINC IO services that read or write data of different size. This form of history dependence is much more difficult to attenuate as the algorithmic behaviour (e.g., application logic) cannot be completely removed, unless we do not force an overly pessimistic constant-time behaviour. We will get back to this issue in the next Section.

An OS layer that meets the above requirements is *time-composable* in that it can be seamlessly composed with the user-level `APPLICATION SW` without affecting its timing behaviour. In the following we present the implementation of a set of `KERNEL`-level services that exhibit a steady timing behaviour and do not disturb the timing behaviour of the user-level code. Our approach seeks for a general reduction in the effects of the OS layer on the application code and is expected to ease the analysis process, regardless of the timing analysis technique of choice.

3 Time-composable kernel layer

So far we reasoned on time composability between the OS and the user application layer. The original POK was not developed with time composability in mind, but rather aimed at the optimisation of the average-case performance. This section describes an alternative design and implementation aimed at injecting time composability in the POK framework. We start our discussion with the basic kernel design choices on time management and system scheduling an then proceed with considerations on some ARINC services we studied. In doing so, we refer to ARINC-specific concepts such as processes, partitions, scheduling slots, etc., whose detailed description is out of the scope of this paper: the interested reader is referred to [1]. Interestingly, similar ideas and solutions can be transposed to different execution platforms.

3.1 Time management

Time management, as one of the core OS services, is exploited by the operating system itself to perform back office activities, and by the application, which may have to program time-triggered actions. Most common time-management approaches adopted in real-time systems rely on either a tick counter or programmable one-shot timers. The original POK implementation provides a tick-based time management where a discrete counter is periodically incremented according to a frequency consistent with the real hardware clock rate[1]. Unfortunately, in tick-based approaches the operations involved in time management are periodically executed, regardless of the application logic; this is likely to incur timing interference on user applications, commensurate to the tick frequency.

For this reason we implemented a less intrusive time management mechanism based on interval timers, where clock interrupts are not necessarily periodic and can be programmed according to the specific application needs. Intuitively, a timer-based implementation can be designed to incur less interference in the timing behaviour of the user application as it guarantees that the execution of a user application is interrupted only when strictly required (i.e., partition switch, process activation, etc.). Making a step further, interval timers also enable to control and possibly postpone timing events at desired points in time and possibly in a way such that user applications are not interrupted. In particular, in an ARINC context we can program timers to expire only at partition switches, so that no overhead is introduced during application execution. Within each scheduling slot we enforce a variant of the fixed-priority deferred scheduling policy [13], in which preemption is enabled only at the end of a job (i.e., *run-to-completion* semantics).

3.2 Scheduling primitives

We implemented a lightweight constant-time – O(1) – fixed-priority scheduler exploiting an extremely compact representation of task states, that can be quickly updated through fixed-latency bitwise operations. In our implementation we assume all processes[2] defined in the same partition to have distinct priorities, to overcome the variability from linear-time insertion in FIFO priority queues. Since hard real-time operating systems typically define 255

[1] POK in its architectural-dependent implementation for PowerPC exploits the decrementer register and the TBU to periodically increment the tick counter.

[2] It should be noted that *process* is the ARINC equivalent of a *task* in classic real-time theory.

distinct priority levels, requiring distinct priorities poses no restriction on ARINC applications which are required to support up to 128 processes per partition [1].

Basically, we exploit a set of bit masks $MASK^{state}$, one for each state a process can assume (i.e., *dormant*, *waiting*, *ready* and *running* in ARINC speak), which collectively describe the current state of all application processes. A similar set of bit masks $MASK^{state}_{slot}$ is associated to each scheduling slot in a major frame, to describe process state changes. State updates are performed by bitwise OR-ing those masks. A simple priority-driven thread selection is done in a similar way by exploiting an ordered bitmask to represent priorities: selecting the runnable process with higher priority thus requires to identify the most significant bit in such mask. Such operation can be performed in constant time with built-in processor instructions (e.g., *count-trailing zeros* on PowerPC) or using perfect hashing with De Bruijn sequences [5].

Process activation events, however, can be dynamically programmed by the user application to occur within a scheduling slot, and thus outside of partition switches. This is the case, for example, when a synchronous kernel service requires a scheduling event to be triggered as a consequence of a timeout[3]. This kind of timeout can be used to enforce, for example, a timed self-suspension (i.e., with "delay until" semantics) or a phased execution of a process. Since we want to ensure that every process is run to completion, preemption is necessarily deferred at the end of process execution, which therefore becomes the next serviceable dispatching point, as shown in Figure 2; dispatching is performed using the same method presented above. A similar mechanism is used for aperiodic processes (i.e., sporadic tasks): in this case, the deferred scheduling event is triggered by a synchronous activation request, which does not involve the use of timers.

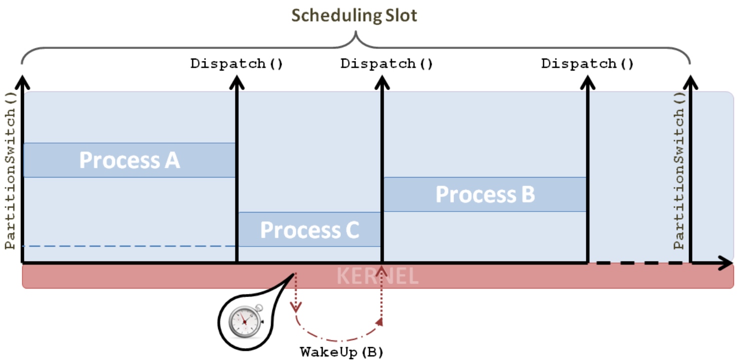


Figure 2 Deferred dispatching mechanism within a time slot.

3.3 Time-composable ARINC APEX

With respect to the subset of ARINC services we have implemented so far, the main timing-composability issues arise from the IO communication between partitions. The basic message-based communication mechanisms provided by the ARINC SAMPLING and QUEUING services is based on channels, as logical links between one source port and one or more destination ports. The timing-composability issues raised by IO services, either through

[3] The DELAYED_START and TIMED_WAIT ARINC services are representative examples of requests for a timed-out process activation.

sampling or queuing ports are mainly due to the variability induced by the amount of data to be read or written. Whereas ports are characterised by a maximum size, forcing the exchange of the maximum amount of data would obtain a constant-time behaviour at the cost of an unacceptable performance loss. Moreover, the potential blocking incurred by queuing port could further complicate the disturbing effects of inter-partition communication. Also the natural countermeasure of isolating the effects of the service execution on the hardware state cannot be seamlessly applied in this context. Inhibiting the caches for example is likely to kill performance since the read and write operations are inherently loop intensive and greatly benefit from both temporal and spatial locality.

To counter this unstable and disturbing behaviour we separate the variable (loop-intensive) part of the read/write services and accommodate such variability so that it incurs less disturbing effects on the execution of the application code. The concrete specification of an ARINC system typically takes a static configuration (e.g., configuration tables) that provides insightful information on the system functional behaviour. We exploit the available information on the inter-partition communication patterns to perform some sort of preventive IO in between partition switch, as depicted in Figure 3.

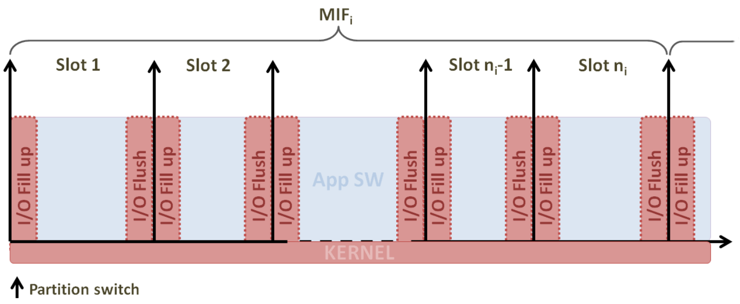


Figure 3 Inter-partition IO management.

We postpone all port writes to the slack time at the end of a partition scheduling slot. Similarly, we preload the required data into the destination partition in a specular slack time, at the beginning of a scheduling slot. The information flow is guaranteed to be preserved as we are dealing with inter-partition communication: (i) the state of all destination (input) ports is already determined at the beginning of a partition slot; (ii) the state of all source (output) ports is not relevant until the partition slot terminates and another partitions gets scheduled for execution. This way, we should not worry about the disturbing effects on the hardware state as no optimistic assumption should ever be made on partition switching; moreover, the input-dependent variability can be analysed within some sort of end-to-end analysis. We are currently implementing a similar approach with respect to intra-partition communication (i.e., via ARINC blackboards, buffers etc.).

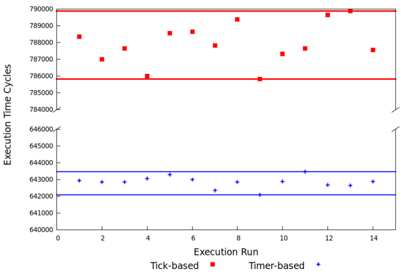
4 Experimental assessment

In our experiments we wanted to assess whether and to what extent our preliminary implementation of kernel primitives and services achieve time composability between OS and user application. We performed our analysis on the basis of the timing information collected by uninterrupted and consecutive end-to-end runs of software units at the granularity level of kernel primitives, ARINC services and application main procedures. Measurements were perfectly suited to meet our objectives as the set of properties we wanted to prove on the OS layer (steady timing behaviour and zero-disturbance) can be arguably assessed by means of a small number of selective examples. In fact, in the absence of history dependence, the timing behaviour of the analysed procedures rapidly fall into predictable behavioural patterns.

The PROARTIS Sim tool, a SocLib based simulator of a PowerPC 750 platform developed within the PROARTIS project, was used to collect timing traces that were later fed to RapiTime [11], a hybrid measurement-based timing analysis tool from Rapita Systems Ltd. The adopted simulator is highly configurable and has been designed to guarantee fixed-latency execution of each processor instruction, except for memory accesses whose latency depends on the current cache state. Since caches are the only residual source of history dependence we were able to exclude, when needed, any source of interference in the execution time by simply enforcing a constant response of the cache, either always miss (i.e., inhibition) or always hit (i.e., perfect cache). The simulator tracing capabilities allowed us to collect execution traces without actual software instrumentation, thus avoiding the so-called *probe effect*. The baseline PROARTIS Sim configuration in our experiments included the perfect cache option, which corresponds to enforcing the latency of a cache hit on every memory access. In the lack of a fine-grained control over the cache behaviour, this parametrisation was meant to exclude the variability stemming from caches without incurring the peformance penalty of thoroughly disabling them. According to our overall approach, in fact, the majority of our experiments address those services that are executed outside of the user application and there is no need to execute them with acceleration features disabled. It is worth noting that the raw numbers obtained under the always hit option are directly proportional to those obtainable under an always miss policy; thus, providing both would not add to our reasoning.

Our experiments were conducted over a relevant set of OS services, which we considered to be the most critical ones from the timing composability standpoint: *time management*, *scheduling primitives*, and *sampling port communication*. All of them were measured under different inputs or different task workloads (i.e., for kernel primitives).

We wanted to first measure whether and to what extent the basic time-management primitives may affect the timing behaviour of a generic user application. We evaluated first the performance of a selective application within the original POK implementation, which uses the decrementer register as a tick counter. Subsequently, we set up a new scenario where no interference arises from the time management service, as the latter was implemented by interval timers set to fire outside the execution boundaries of the examined procedure. Caches have been enabled for this experiment and configured with Least Recently Used (LRU) replacement policy.



	MinET	MaxET	Delta
Tick-based	785825	789888	4063
Timer-based	642076	643472	1396

Figure 4 Execution under tick-based and interval-timer time management.

The experimental results shown in Figure 4 are not surprising. The tick-based time management mechanism (upper band in the plot) should be discarded in favour of the interval timer, since its disturbance on the application code is clearly higher, due to the set of useless time-management activities performed on every tick. Interestingly, as highlighted by the different areas between the straight lines, the cache-induced variability experienced by the application under a tick-counter policy is considerably greater than that suffered under interval-based timer, as a consequence of increased pollution of cache states.

Moving on to scheduling primitives, we observe that inattentive implementation and design choices may affect both the latency and jitter incurred by scheduling primitives such as partition switch, process dispatching or state update. To provide experimental evidence of the steady timing behaviour of our implemented scheduling primitives, as opposed to the standard ones in the original version of POK, we focus on task status update and task election. These activities are performed in a single operation in tick-based approaches, whereas they execute separately in our approach: this is because status update is performed only at partition switch, whereas thread dispatching occurs at the end of every job execution, according to the run-to-completion semantics. We enforced a perfect cache behaviour so that no overhead from the hardware is accounted for in measured execution times. We also concocted our experiments to follow a strictly deterministic periodic pattern, which allowed us to restrain our observations to a limited number of runs. Figure 5 shows observed execution times for the thread selection routine (that is part of the larger scheduling primitive). The workload in the top chart is two partitions, with three and two threads respectively, while the bottom chart reports a larger example comprises three partitions with ten, five and two threads each.

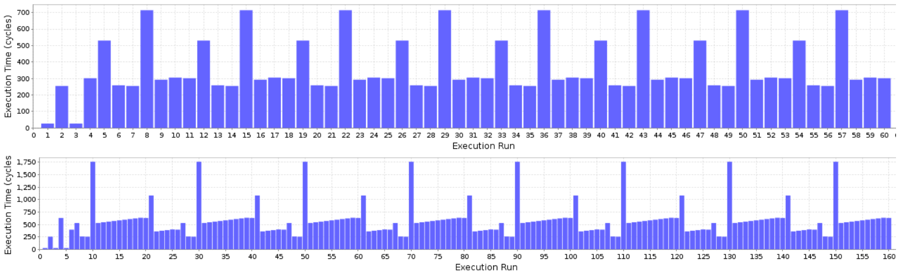


Figure 5 FPPS thread selection under different workloads.

The original POK scheduler always performs the same operations at every clock tick, mainly checking whether the current partition or thread should be switched to the next ones. Under those premises, two potential sources of variability originate from the possibility that a partition/thread needs to be actually switched at a particular scheduling point, and from the number of threads to be managed in the executing partition, respectively. The graphs in Figure 5 illustrate this situation clearly: higher peaks correspond to partition switches, when the state of all threads in the new partition changes to ready and they must be therefore inserted in the appropriate scheduler queues. For our constant-time scheduler, instead, we must distinguish two cases, since its behaviour is different at partition and thread switch. Figure 6 shows the execution time of the routine invoked at partition switch, which only needs to update thread states[4]. Though the settings are exactly the same as Figure 5 above, status updates are performed in constant time thanks to the bitwise operations on thread masks (Section 3.2).

Figure 7 shows that our constant-time scheduler is capable of detecting the highest-priority thread to be dispatched with fixed overhead, by using De Bruijn sequences. Lower

[4] Except for inter-partition communication overhead.

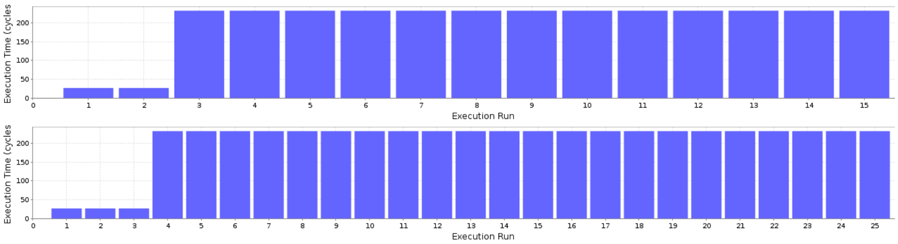


Figure 6 Constant-time thread status update under different workloads.

peaks in Figure 6 correspond to the selection of the system idle thread. From the raw numbers, reported in Table 1, we note that the small delta exhibited by our thread switch implementation is actually due to the difference between the selection of any thread (95) and the idle thread[5] (45). The delta measured on the standard POK implementation, instead, represents real jitter.

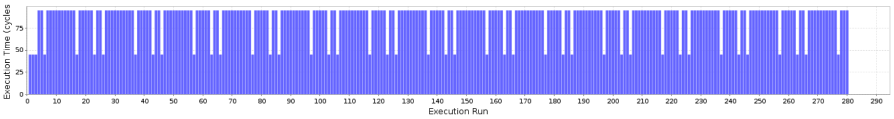


Figure 7 Constant-time thread selection in a test case with three partitions and seventeen threads.

Table 1 Execution times for a user application with tick-based and interval-timer scheduling.

	FPPS (standard POK)			O(1) scheduler (partition switch)			O(1) scheduler (thread switch)		
	Min	Max	Delta	Min	Max	Delta	Min	Max	Delta
2 partitions 5 threads	255	714	459	232	232	0	45	95	50
3 partitions 17 threads	259	1759	1500	232	232	0	45	95	50

When it comes to ARINC APEX, we focused on inter-partition communication to start with. Inter-partition communication via sampling ports have been specifically redesigned for the sake of time composability[6]: our implementation is based on *posted* writes and *prefetched* reads that permits to remove the sources of variability and disturbance from the service itself and serve them out at partition switch. We forced the simulator to resemble a perfect cache when measuring this services as we wanted to exclude the variability stemming from the cache behaviour without incurring the peformance penalty of a disabled cache: a positive effect of relocating the message management in between the execution of two partitions is in fact that of being able to exploit the caches without any side-effect on the user application. Having excluded the cache variability causes the analysed service to exhibit a steady timing (actually constant) behaviour where the execution time only varies as a function over the input (message) size. We triggered the execution of the sampling services with different sizes of the data to be exchanged. The dependence of READ and WRITE services on the input size

[5] The idle task is elected for execution when no other task is runnable.
[6] The implementation of a similar mechanism for queuing ports is work in progress at the time of writing.

is shown in Table 2: the increase in the execution time of each service is related to an increase in the input data size, here ranging from 1 to 256 Bytes. The redesigned implementations of both services (newWRITE and newREAD in Table 2) are instead constant, as the invocation of the services themselves does actually execute neither a read nor a write operation, whose execution is instead deferred at the begin and end of a partition switch respectively.

Table 3 shows the partition switch overhead (observed under different input sizes) that is the penalty that has to be paid for relocating the message passing mechanism on partition switch. From what we observed in our experiments, the incurred time penalty is quite limited and, more importantly, when summed to the time previously spent in the READ or WRITE service, it does not exceed the execution time of the standard implementation with the same input.

Table 2 Execution times for the READ and WRITE services.

	WRITE	NewWRITE	READ	NewREAD
1B	523	436	618	448
4B	580	436	794	448
32B	1112	436	1383	448
64B	1720	436	1758	448
96B	2024	436	2086	448
128B	2936	436	2974	448
256B	5368	436	5406	448

Table 3 Maximum observed partition switch overhead.

	Partition Switch	Read+Write
	(standard)	Overhead
32 B	27674	+ 661
64 B	29498	+ 1269
96 B	32224	+ 1973
128 B	33146	+ 2485
192 B	37686	+ 3807
256 B	41619	+ 5118
384 B	48630	+ 7455

5 Conclusion

Composability in the time dimension is a fundamental enabler for the hierarchical decomposition of large complex systems into smaller, tractable units. Whereas hardware platform are widely acknowledged to have great influence on the timing composability, in this paper we focus the role of the real-time operating system in enabling timing composability in IMA systems and identified the properties that make an operating system timing-composable with user applications. In that light, we redesigned a real-time partitioned kernel and provided experimental evidence that the degree of time composability may greatly benefit from proper design choices in the implementation of the operating system.

References

1 APEX Working Group. Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface. 2003.

2 Julien Delange and Laurent Lec. POK, an ARINC653-compliant operating system released under the BSD license. *13th Real-Time Linux Workshop*, 10 2011.

3 F.J. Cazorla et al. PROARTIS: Probabilistically analysable real-time systems. *ACM Transactions on Embedded Computing Systems*, to appear.

4 Freescale. PowerPC 750 Microprocessor, 2012. `https://www-01.ibm.com/chips/techlib/`
`techlib.nsf/products/PowerPC_750_Microprocessor`.

5 Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de Bruijn Sequences to Index a 1 in a Computer Word, 1998.

6 Isaac Liu, Jan Reineke, and Edward A. Lee. A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties. In *44th Asilomar Conference on Signals, Systems, and Computers*, pages 2111–2115, November 2010.

7 Ingo Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler, Jan. 2002. Available on-line at `http://casper.berkeley.edu/`, visited on April 2012.

8 F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

9 Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time software. In *Proc. 1st International Workshop on Software Technologies for Future Dependable Distributed Systems*, Mar. 2009.

10 Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis. In *Proc. of the 8th Int. Workshop on WCET Analysis*, 2008.

11 Rapita Systems Ltd. Rapitime, 2012. `http://www.rapitasystems.com/rapitime`.

12 J. Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

13 Gang Yao, Giorgio C. Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with limited preemptions. *Real-Time Systems*, 47(3):198–223, 2011.