A Time-Composable Operating System for the Patmos Processor

Marco Ziccardi
Department of Mathematics
University of Padua
mziccard@math.unipd.it

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk

Tullio Vardanega
Department of Mathematics
University of Padua
tullio.vardanega@math.unipd.it

ABSTRACT

In the last couple of decades we have witnessed a steady growth in the complexity and widespread of real-time systems. In order to master the rising complexity in the timing behaviour of those systems, rightful attention has been given to the development of time-predictable computer architectures. The Patmos time-predictable microprocessor used in the T-CREST project employs performance-enhancing hardware while keeping the system analyzable. Time composability, at both hardware and software level, is a considerable aid to reducing the integration costs of complex applications. A time-composable operating system, on top of a time-composable processor, facilitates incremental development, which is highly desirable for industry. This paper makes a twofold contribution. First, we present enhancements to the Patmos processor to allow achieving time composability at the operating system level. Second, we extend an existing time-composable operating system, TiCOS, to make best use of advanced Patmos hardware features in the pursuit of time composability.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.4.1 [**Operating Systems**]: Process Management; D.4.8 [**Operating Systems**]: Performance

Keywords

Real-time Operating System, Time Composability, Time Predictability

1. INTRODUCTION

Real-time systems pervade our everyday life. Their use span multiple industrial sectors, including industrial applications, automotive, aerospace, and avionics. The processing demand of modern real-time systems grows steadily: the number of applications running on the same system as well as their need for computational resources keep increasing. The complexity growth of real-time applications is visible across the entire execution stack: (1) performance-enhancing hardware (e.g., caches) is employed to enable faster execution; (2) specialized real-time operating systems (RTOS) are used to efficiently handle resource sharing. The correctness of a real-time system must be assured in the functional and time domains. In order to assess that a certain operation will always complete within a certain time limit, timing analysis is used to provide an upper bound of the worst-case execution time (WCET). Some hardware resources employed by modern architectures exploit information on the execution history to improve average case performance. Such a history-dependent behavior considerably increases the complexity of timing analysis techniques. Recent research efforts have addressed the development of time-predictable processor architectures (e.g., T-CREST [16][1] and MERASA [18]) to make the worst-case fast while keeping system behavior easy to analyze [14].

The ability to sustain incremental development for modern real-time systems is becoming a crucial asset to contain the growing design, development and validation costs. Incrementality, however, is only be granted when the application's timing behavior stays unchanged in the face of system integration. In other words, the system should behave in a composable manner in the time dimension (thus being a time-composable system) [12]. The processor resources should not exhibit large time jitter as a result of history dependence or variations in the intensity of contention. Similarly, RTOS services must be designed to ensure that the analysis results computed for a program on a time-predictable hardware platform continue to hold upon system integration. That is, the RTOS must be time-composable [7]. In [6], two properties of such a RTOS are identified: (1) *zero disturbance* and (2) *steady timing behavior*. Zero disturbance requires that the RTOS should not affect the timing behavior of applications by altering the state of jittery (state-dependent) hardware resources. Steady timing behavior, instead, avoids significant variability in the execution time of the operating system's services. To be time-composable, these services have to be designed to always take (near) constant time. To sustain their claim, the authors of [6] present TiCOS, an open-source time-composable RTOS conforming to the ARINC 653 standard.

Our work here makes two complementary contributions. First, we identify the minimum set of hardware features

[1]Time-predictable Multi-Core Architecture for Embedded Systems (`http://www.t-crest.org/`)

needed by a time-predictable architecture to support the execution of a multi-threaded RTOS. Such features have been investigated and implemented in the Patmos processor. Second, we studied how hardware accelerators employed by a time-predictable processor may impact the time composability of an operating system. Means to exploit the increased performance provided by such a processor, without negatively affecting time composability, have been explored and developed in TiCOS. Some accelerators, as scratchpad memories, have also been shown to facilitate the development of constant-time services as well as to increase performance.

This paper is organized as follows: Sections 2 describes background information on the Patmos processor and the TiCOS operating system. Section 3 introduces some related work. Section 4 presents extensions applied to the Patmos processor while Section 5 details changes performed on the TiCOS architecture. Section 6 presents our experimental setups and the results obtained. Finally, Section 7 summarizes our work and underlines its main contributions.

2. BACKGROUND

In this section we provide background knowledge on the Patmos processor and the TiCOS operating system.

2.1 The Patmos Processor

Patmos [15] is a time-predictable, RISC processor for use in real-time systems. Patmos features 32 general-purpose and 16 special-purpose registers. The first special-purpose register holds 8 1-bit predicate registers used for predicated instructions. Local fast memories hold stack, data, and instructions to enhance performance while keeping the WCET analysis simple. Besides a write-through data cache and two scratchpad memories for instructions and data, also a method cache and a stack cache are provided.

The method cache [13] is a special instance of an instruction cache: the replacement mechanism operates on function calls and returns. When a function is called all its corresponding blocks are fetched into the cache. When a function returns the blocks containing the code of the caller are, if needed, fetched again.

The stack cache [4] is a dedicated cache meant to hold stack allocated values. Data allocated on the stack is accessed very frequently and notably benefits from caching. A stack frame allocated on the stack cache does not necessarily have to be consistent with the main memory: once a function returns its stack frame can be discarded. That is, stack cache consistency is, in general, not needed. In some situations, however, when the number of nested function gets large enough, some stack frames might be saved (spilled) to memory to make room for more frames. When the functions return, frames saved in main memory can be restored. The stack cache is managed by the compiler via three instructions in Patmos:

`sres`: on a function call, a stack frame of the required size is reserved on the top of the stack and in the stack cache

`sens`: after a function returns, ensures the caller's frame to be in the stack cache (it may have been spilled to the main memory)

`sfree`: pops the frame from the top of the stack and from the stack cache.

The Patmos stack cache is placed in the local memory and managed through 2 pointers saved to special-purpose registers: (1) `st` points to the top of the stack; (2) `ss` points to the last element of the stack cache spilled to main memory. In order to allocate aliased data, a secondary, non-cached stack called *shadow stack* is used by the compiler.

2.2 The TiCOS Operating System

TiCOS [6] is a light-weight RTOS, derivative of POK [9], and conforming with ARINC 653. The kernel has been modified and further extended to attain time composability. In keeping with the ARINC 653 standard, the TiCOS kernel supports partitions, the ARINC jargon for applications, each of which can include one or more processes (a thread in ARINC speak). Partitions are time and space isolated. Space isolation is obtained by granting to each partition its own memory and disallowing shared memory. Time isolation is attained by assigning to each partition a scheduling slot. A two-level scheduling algorithm is provided: partitions are selected out of a static scheduling table; the processes of an active partition are scheduled in fixed-priority order. Servicing in constant time removes jitter from the time behavior of the RTOS. The TiCOS scheduler also enforces *run-to-completion* semantics: a process can become ready at any time of in the partition's scheduling slot, but its activation is deferred until the end of the currently running process. That is, jobs issued by a process are never preempted by other processes in the same partition.

Processes in a partition are allowed to synchronize via ARINC 653 events. Events can be in one of two states: *set* or *reset*. When an event is in the *set* state it allows all the waiting processes to continue. When an event is in the *reset* state all processes waiting for the event are blocked [17].

Inter-partition communication is enabled through ARINC 653 ports. Since partitions do not share memory, the kernel is responsible for copying messages from source ports to destination ports. Ports enable message passing and can be a threat to time composability due to variability in the size of exchanged data. To prevent jittery behavior inside partitions, the data-dependent part of all port operations is placed in the *slack* time between scheduling slots.

3. RELATED WORK

A number of operating systems, for example PikeOS [3], INTEGRITY [1], LynxOS-178 [2], and XtratuM [11] isolate applications in time and space and provide support for the ARINC 653 standard. However, isolation is implemented in different ways. PikeOS, LynxOS-178 and XtratuM isolate applications through the adoption of virtual machines. PikeOS runs different applications (hard real-time, soft real-time, and non-real-time) on different virtual machines. Scheduling is both priority- and time-driven so that computing time not used by hard real-time tasks can be re-allocated to non-real-time ones. LynxOS-178 divides time in fixed-size slices according to ARINC 653 specification. Memory isolation is guaranteed by dividing the address space in blocks, each of which is assigned to a partition. PikeOS, LynxOS-178 and XtratuM also enforce resource partitioning so that every resource is associated to one and only one partition at any time. By doing this, a fault can be handled in a single partition, without affecting the others. However, while providing a more effective way to isolate applications (also interrupts can be virtualized), virtualization comes at the cost of more complexity at the kernel services level, thus challenging time composability.

Similarly to TiCOS, INTEGRITY does not employ virtual machines. Time isolation is realized in it through the ARINC 653 two-level scheduling. Space isolation is implemented using statically configured Memory Management Units (MMU). MMUs configuration is never changed and partitions are not allowed to share memory. INTEGRITY kernel services always use resources owned by the calling partition, no new objects are placed in memory to perform system calls. As in TiCOS, OS services are executed on a dedicated stack to prevent stack overflow and to allow user processes to precisely specify their stack size. INTEGRITY, as TiCOS, avoids recursion in kernel services so that kernel stack size can be statically determined. While TiCOS places message transfers for inter-partition communication in the slack time between partitions, INTEGRITY uses partition time to that end.

None of PikeOS, INTEGRITY or LynxOS-178 ensures time composability of kernel services although they all provide isolation and support the ARINC specification. Conversely, CompOSe [10] is a composable operating system targeting Multi-Processor Systems on Chip (MPSoC). Similarly to TiCOS, CompOSe employs a two-level scheduler to have different intra-application policies and exploits the slack time for inter-application communication. However, CompOSe does not support the ARINC standard.

4. PATMOS EXTENSIONS

In order to support time sharing of the processor and the execution of an operating system, Patmos has been extended with interrupts and stack cache manipulation instructions. Other improvements, e.g., memory protection, are being investigated as part of current research.

4.1 Interrupts and RTC

Interrupts provide means for reacting to external events by stopping the current flow of control and jumping to a dedicated Interrupt Service Routine (ISR). Interrupts are used by a RTOS to manage time and perform scheduling. Usually, time management is performed via a tick counter or programmable one-shot timer [6]. When a tick counter is used, interrupts are periodically triggered and time management routines are activated. This recurrent activation may cause such routines to interfere with user applications. In order to reduce this interference, a time-composable operating system should prefer interval timers to tick counters. An interval timer can in fact be programmed to expire only when required (e.g. partition switches). Modern architectures provide ways to program timers to expire at specified intervals. In order to have interval interrupts the Patmos processor has been extended with a memory-mapped *Real-Time Clock* (RTC). The Patmos RTC maps different registers to the local memory, as shown in Table 1. The interval timer can be set by writing a value either to the *Clock cycles* registers or to the *Time in microseconds* registers.

Interrupts are not only used for time management but also to react to I/O events, to implement system calls, and to handle exception states. In order to enable configuration and handling of several types of interrupts and exceptions Patmos has been further extended with a memory mapped *Exception Unit* that supports a 32 entry *exception vector* to specify ISR addresses. A `trap` instruction can be used by an operating system to generate internal exceptions (e.g., for system calls). The `trap` instruction takes an immediate operand and puts the processor in the corresponding exception state. When an exception occurs, regardless of its type, the return address is saved to a special purpose register to be subsequently used by the `xret` instruction to return from the exception state to the interrupted control flow.

Address	I/O Device
`0xf0000200`	Clock cycles (lower 32 bits)
`0xf0000204`	Clock cycles (upper 32 bits)
`0xf0000208`	Time in microseconds (lower 32 bits)
`0xf000020C`	Time in microseconds (upper 32 bits)

Table 1: Memory mapped RTC registers

4.2 Stack Cache Manipulation

In a multi-threaded environment, context switching is used to interleave parallel executions. On a context switch, the state of the processor (context) for the currently executing thread is saved. Then, the context of some other thread has to be restored. In Patmos, the state of the stack cache is part of the context of a thread; as such, it has to be stored before and restored after a context switch.

To store a possibly inconsistent stack cache state (as the one shown in Figure 1a) a new processor instruction, `sspill`, has been added to Patmos. `sspill` takes an immediate or a register parameter and saves the specified amount of stack cache in main memory downward, starting from address `ss`. The amount of stack stored in the stack cache and not persisted in main memory yet can be easily computed as the difference between the stack spill pointer `ss` and the stack top pointer `st`. Figure 1 shows the execution of the `sspill` instruction on a non-consistent cache state. Figure 1b shows the state of the cache after `sspill` is executed.

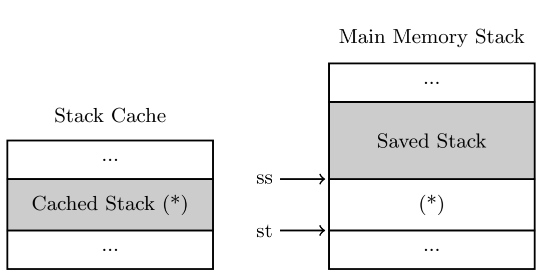
To restore a context, also the state of the stack cache has to be restored. To this end, the `sens` instruction has been modified to accept a register parameter `ssize`. With `ssize = ss - st` denoting the amount of thread's space in the the stack cache at the point of interruption, executing `sens ssize` restores the state of the cache before interruption. `ssize` is always stored as part of the context.

5. TICOS EXTENSIONS

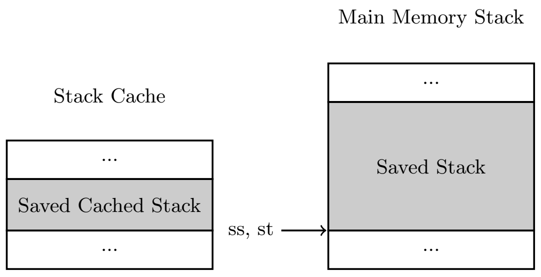
The TiCOS kernel is divided into two layers: *arch* and *core*. While the *arch* layer contains all architecture-dependent functionalities the *core* layer includes all kernel's basic functionalities that are used to implement ARINC services. ARINC 653 processes, for instance, are implemented using TiCOS *core* threads. The adaptation of the operating system for the Patmos processor followed an incremental fashion. First, the architecture-dependent layer has been developed and tested. Then, the core functionalities have been adapted to the Patmos architecture. Incrementality enabled a faster and less error prone development of the *core* layer. Main changes to the *arch* layer regarded the definition of a thread's context and memory management. On the other hand, in the *core* layer, the context switch mechanism has been modified to handle Patmos local memories and the bootloader has been extended to gain more flexibility.

5.1 Thread's Context

The context of a thread in Patmos is made of the 32 general-purpose registers, the 16 special-purpose registers, the state of the exception unit and the state of the stack



(a) Stack cache inconsistency during execution



(b) Stack cache configuration after `sspill` instruction

Figure 1: Stack cache manipulation through the `sspill` instruction

cache. A C data structure is used to store all that state information, along with the size of the cached stack `ssize`.

5.2 Memory Management

In TiCOS, each ARINC 653 partition is defined as a main function responsible for creating all partition's processes, each of which is mapped to a TiCOS thread. In order to initialize a partition, a main thread whose entry point is the partition's main function is created. The OS also employs an idle thread, scheduled for execution when no other user thread is ready. The Patmos compiler uses two types of stacks, a cached stack and a shadow stack. To work properly, each thread has to be granted space for both stacks.

TiCOS is designed so that the maximum size of the stack can be computed, for each thread, at development time. System calls are, in fact, executed on dedicated kernel's stacks. In the absence of recursion the size of the stacks for user threads can be therefore statically determined. At development time, the user specifies the amount of memory each partition will require. In this memory area stacks (cached and shadow) for all partition's threads, communication buffers and ports are allocated. At system startup, a cached and a shadow stack are also allocated for both operating system calls and the idle thread. Threads contexts are hold in a dedicated operating system data structure. A pointer `current_context` always points to the entry in the data structure holding the context of the currently running thread. `current_context` is used by scheduling routines to save and restore execution contexts.

5.3 Bootloader

An ARINC 653 system is made of several applications, each assigned to a partition. TiCOS applications are compiled against the OS library into an independent executable file. At system startup the operating system loads partitions and, once loaded, retrieves partitions entry points to initialize and start main threads. The process of loading partitions executables can be carried out in several ways. Two loaders have been implemented in TiCOS: a *static bootloader* and a *dynamic bootloader*.

When the static bootloader is used, partitions are compiled first. The build script then compiles the kernel providing it with an array containing partitions entry points. Finally, partitions are placed inside kernel's executable. The static bootloader is effective and fast. In fact, when partitions are initialized by the OS they have already been loaded in memory. However, effectiveness and speed detract from flexibility: small changes in partitions code force the user to re-compile both partitions and kernel.

The dynamic bootloader instead leverages Patmos I/O to obtain more flexibility. The Patmos processor is equipped with a memory-mapped Universal Asynchronous Receiver-Transmitter (UART). The dynamic bootloader reads each partition from the UART and places it at proper location in memory. An executable file written in ELF [8] is made of a number of segments described in the Program Header Table. Segments represent information needed for runtime execution of the program. In particular, segments of type `PT_LOAD` correspond to `.text` and `.data` sections of the ELF file and have to be loaded into memory. The stream format expected by the bootloader for each partition contains first the partition's entry point and immediately after the number of loadable segments. Each segment is described by its address, size and data. Since no memory virtualization is implemented in Patmos, the address of a segment specifies where the loader has to place it into main memory.

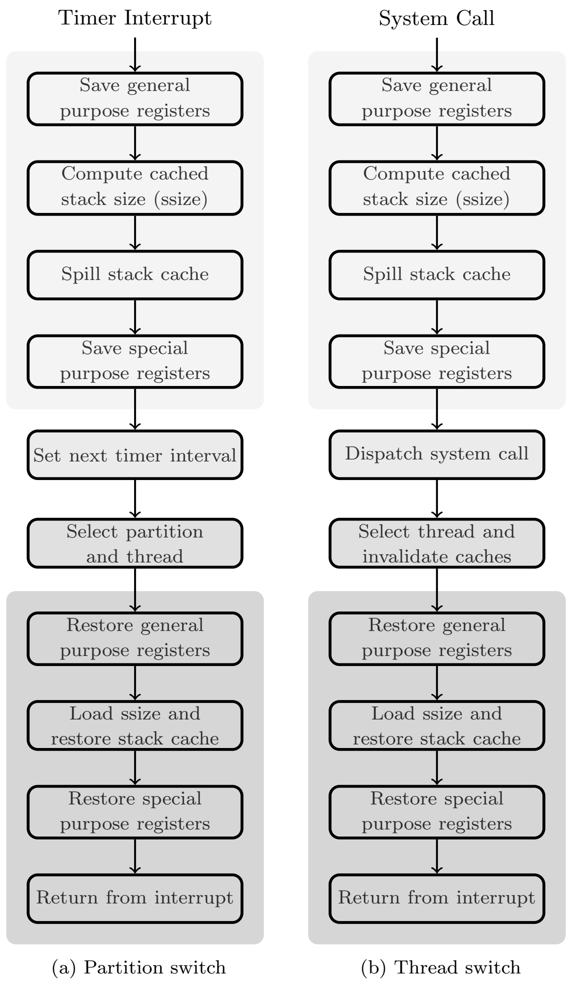
5.4 Context Switch

The context switch mechanism is designed to be as light weight and simple as possible. Thread contexts are held in a OS data structure and `current_context` pointer always indicates the entry corresponding to the currently active thread. When an interrupt occurs, either for I/O, timer expire or system call, `current_context` is used to save the current state of the processor. Conversely, after performing scheduling decisions, `current_context` points to the thread elected for execution. TiCOS enforces run-to-completion semantics and employs a two level scheduling. First-level scheduling occurs between partitions at each time slot end and is activated by the timer interrupt. Second-level scheduling arbitrates between threads inside a partition. Owing to run-to-completion, thread switch within a partition is activated by an explicit system call. We can indeed distinguish between two types of context switches: *partition switch* and *thread switch*, as shown in Figure 2.

5.4.1 Partition Switch

When a partition slot ends a timer interrupt is raised and the control flow is transferred to a dedicated interrupt service routine. `current_context` points to the memory area where to save the context of the currently executing thread. After saving the context the timer interval is set to expire at the next time slot end. Once the timer interval has been set, a scheduling decision has to be performed: a new partition and a thread inside it have to be selected for execution. Eventually, the context of the newly selected thread is restored. The interrupt-driven context switch procedure is outlined in Figure 2a.

Partitions are space isolated and represent different unit of analysis. Partition switch can indeed exhibit a variable



Timer Interrupt
Save general purpose registers
Compute cached stack size (ssize)
Spill stack cache
Save special purpose registers
Set next timer interval
Select partition and thread
Restore general purpose registers
Load ssize and restore stack cache
Restore special purpose registers
Return from interrupt
(a) Partition switch
System Call
Save general purpose registers
Compute cached stack size (ssize)
Spill stack cache
Save special purpose registers
Dispatch system call
Select thread and invalidate caches
Restore general purpose registers
Load ssize and restore stack cache
Restore special purpose registers
Return from interrupt
(b) Thread switch

Figure 2: Control flow of a context switch

timing behavior without affecting results computed in the analysis. Therefore, the data-dependent part of all port operations (that can exhibit variable latency) is performed at partition switches.

5.4.2 Thread Switch

A thread switch inside a partition is explicitly requested, for instance, at the time of the periodic release of an ARINC 653 process. Explicit context switch can only be requested by kernel services. When a system call is executed, the context of the caller is saved. After saving the context the kernel service corresponding to the system call is executed. The kernel service might, possibly, invalidate data and method cache and select a new thread inside current partition to be executed. That is, the location pointed by `current_context` might change. Once the kernel service is executed the context has to be restored. Restoring the context, if `current_context` changed, causes the need to restore the context of a new thread. The thread switch procedure, starting from the system call, is shown in Figure 2b.

6. EVALUATION

We performed a number of experiments to assess whether the new implementation of TiCOS for the Patmos processor preserved time composability between kernel services and applications. In particular, we wanted to provide experimental evidence that the steady timing behavior property of scheduling primitives is preserved. Measurements for explicit context switch have been collected. While in [6] thread selection and thread status update routines were analyzed, in our experiments we measured end-to-end thread switches (from system call interrupt to the return from interrupt) to ascertain whether the context switch time holds constant.

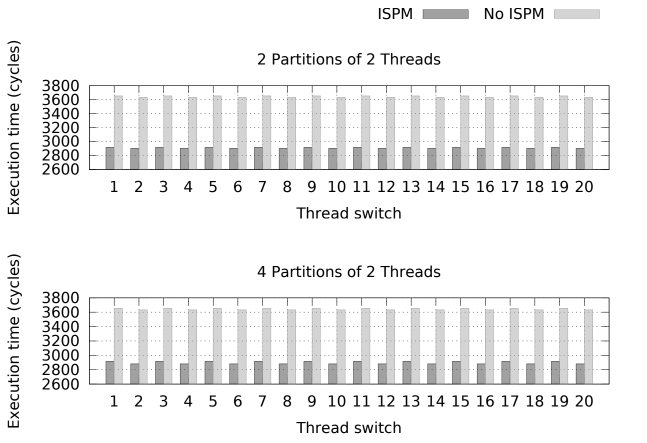


Figure 3: Constant-time thread switch under various workloads

Patmos is implemented in Chisel [5], a hardware-construction language developed at the UC Berkeley and embedded in the Scala programming language. The Chisel back-end can generate both Verilog and C++ code. The C++ code implements a fast cycle-accurate simulator of the hardware. The Chisel simulator of Patmos has been modified to be able to collect traces without instrumenting kernel or application code, thus not affecting the timing behavior of either the kernel or the application.

We recall that TiCOS employs an idle thread, a main thread for each partition and a thread for each ARINC process. Experimental results have been collected under different workloads (in parentheses the total number of threads created is specified): *2 partitions of 2 processes* (7 threads) and *4 partitions of 2 processes* (13 threads). Experiments have also been designed to exhibit a strictly periodic behavior so that a limited number of runs is enough to provide experimental evidence of steady timing behavior.

Patmos contains a 1 KB instruction scratchpad memory (ISPM). TiCOS can be configured to use the ISPM by placing kernel routines in it, thus potentially reducing kernel services execution times. In the experimental setup the stack cache has been disabled. The `sspill` instruction, in fact, may introduce variability. To attain time-composability at the kernel level `sspill` should be enforced to always take constant time.

In the experimental setups, partitions contain 2 threads. The first thread switch occurs between two user threads while the second context switch selects the idle thread for execution. Figure 3 shows that the first thread switch in a partition is more expensive than the second one. That is, switching to a user thread in the partition costs more than switching to the idle thread since it requires updating the next thread's deadline. Furthermore, Figure 3 shows that switching to a partition's thread, as well as switching to the idle thread, holds constant across partitions and across ex-

	No ISPM	ISPM	Gain
Switch to Thread	3652	2916	20.1%
Switch to Idle	3632	2900	20.1%

Table 2: Constant time thread switch in clock cycles for both ISPM enabled and disabled

perimental setups. Without using the ISPM, switching to a thread inside the partition takes 3652 clock cycles while switching to the idle thread costs 3632 cycles. When the ISPM is used instead, switching to a thread takes 2916 clock cycles while switching to the idle thread costs 2900 cycles. Experimental results for thread switch are summarized in Table 2. Thread switch costs 20.1% less when ISPM is used. However, as the current ISPM cannot hold all thread switch code, increasing its size might allow for a further performance increase.

7. CONCLUSIONS

Real-time applications are increasing in complexity. The costs of development and qualification grow with the rise in complexity. Time-predictable architectures provide performance-enhancing features while maintaining the system analyzable. Development costs can be lowered by building the system incrementally. Incremental development needs a time-composable execution platform to build on, so that the timing behaviour of an application verified in isolation is not jeopardized by integration with other software.

In this paper we presented the adaptation of the TiCOS time-composable operating system to the Patmos processor. Support for time-predictable hardware features, such as the stack cache, has been provided in TiCOS, identifying and addressing challenges to time-composability. This paper also presented hardware features (e.g., interrupts) that have been introduced in Patmos to support the execution of a time-composable OS.

The results show that steady timing behavior has been preserved for what concerns scheduling primitives. Moreover, this paper shows how the adoption of an instruction scratchpad may increase the application performance by reducing context switch cost. ISPM, in fact, guarantees fast instruction fetch without challenging time-composability.

Source Access

The code of the operating system, the processor, the processor simulator, and the compiler are open source and available at: `https://github.com/t-crest`.

8. REFERENCES

[1] INTEGRITY. `www.ghs.com/products/rtos/integrity.html`.

[2] LynxOS-178. `www.lynuxworks.com/rtos/rtos-178.php`.

[3] PikeOS. `www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/`.

[4] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: constructing hardware in a scala embedded language. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *DAC*, pages 1216–1225. ACM, 2012.

[6] A. Baldovin, E. Mezzetti, and T. Vardanega. A time-composable operating system. In *WCET*, pages 69–80, 2012.

[7] A. Baldovin, E. Mezzetti, and T. Vardanega. Towards a time-composable operating system. In *Ada-Europe*, pages 143–160, 2013.

[8] T. T. I. S. Commitee. Executable and Linking Format (ELF) Specification. `http://pdos.csail.mit.edu/6.828/2012/readings/elf.pdf`, May 1995.

[9] J. Delange and L. Lec. Pok, an arinc653-compliant operating system released under the bsd license. *13th Real-Time Linux Workshop*, 10 2011.

[10] A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, and K. Goossens. Design and implementation of an operating system for composable processor sharing. *Microprocess. Microsyst.*, 35(2):246–260, Mar. 2011.

[11] M. Masmano, I. Ripoll, A. Crespo, J. Metge, and P. Arberet. XtratuM: An open source hypervisor for TSP embedded systems in aerospace. In *DASIA 2009. DAta Systems In Aerospace.*, May. Istanbul 2009.

[12] P. Puschner and M. Schoeberl. On composable system timing, task timing, and wcet analysis. In *In International Workshop on Worst-Case Execution Time Analysis*, 2008.

[13] M. Schoeberl. A time predictable instruction cache for a java processor. In *In On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), volume 3292 of LNCS*, pages 371–382. Springer, 2004.

[14] M. Schoeberl. Time-predictable computer architecture. *EURASIP J. Embedded Syst.*, 2009:2:1–2:17, Jan. 2009.

[15] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[16] M. Schoeberl, C. Silva, and A. Rocha. T-CREST: A time-predictable multi-core platform for aerospace applications. In *Proceedings of Data Systems In Aerospace (DASIA 2014)*, Warsaw, Poland, June 2014.

[17] S. Thompson, G. P. Brat, and A. Venet. Software model checking of arinc-653 flight code with mcp. In *NASA Formal Methods*, pages 171–181, 2010.

[18] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, and J. Wolf. Merasa: Multi-core execution of hard real-time applications supporting analysability. *Micro, IEEE*, 30(5):66–75, 2010.