D2.12 Evaluation report on research-oriented software-only multicore solutions Version 1.0

Document Information

Contract Number	611085
Project Website	www.proxima-project.eu
Contractual Deadline	m36, 31-September-2016
Dissemination Level	PU
Nature	R
Authors	Leonidas Kosmidis (BSC), Tullio Vardanega (UPD)
Contributors	Enrico Mezzetti (UPD/BSC), Davide Compagnin (UPD), Luca Bonato (UPD), David Morales (BSC), Eduardo Quinones (BSC)
Reviewer	
Keywords	RTOS, Timing Composability, Software Randomisation, Multicore, PTA

Notices:
The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 611085.

Change Log

Version	Description of change
v0.1	Initial Draft for internal review
v0.2	Internal review comments addressed

Contents

Executive Summary **5**

1 Introduction **6**

2 Research-oriented RTOSes **8**
2.1 FPGA COTS . . . 10
2.1.1 Task Manager . . . 11
2.1.2 Evaluation . . . 12
2.1.3 Semaphore Manager . . . 13
2.1.4 Evaluation . . . 14
2.1.5 Event Manager . . . 19
2.1.6 Evaluation (limited to per-core events) . . . 21
2.1.7 Utilities . . . 23
2.2 AURIX . . . 26
2.2.1 System Configurations . . . 26
2.2.2 Cyclic Tasks . . . 28
2.2.3 Sporadic Tasks . . . 29
2.2.4 Time-Composable Implementation . . . 30
2.2.5 Evaluation by PAKs . . . 33
2.2.6 Using Erika-TC . . . 37

3 Software Randomisation **39**
3.1 FPGA COTS . . . 40
3.1.1 High Level Description of Dynamic Software Randomisation Port at m18 . . . 40
3.1.2 Limitations of Dynamic Software Randomisation Port at m18 and their solutions . . . 41
3.1.2.1 Relocation Scheme: . . . 41
3.1.2.2 Dynamic Memory Consumption . . . 43
3.1.2.3 Random Cache line Selection: . . . 43
3.1.3 Evaluation . . . 44
3.1.3.1 Cache line Randomisation . . . 44
3.1.3.2 Memory overheads . . . 49
3.1.3.3 Eager vs Lazy relocation . . . 52
3.1.3.4 pWCET Estimations . . . 54
3.1.3.5 Average Performance . . . 54
3.1.3.6 Event distributions in isolation . . . 55
3.2 P4080 . . . 58
3.2.1 Porting details . . . 58
3.2.2 Evaluation . . . 58
3.2.2.1 Cache line Randomisation . . . 59
3.2.2.2 Memory Overheads . . . 62
3.2.2.3 pWCET Estimations . . . 64
3.2.2.4 Event distributions . . . 70

3.3 AURIX 74
3.3.1 High Level Description of Static Software Randomisation/-TASA at m18 74
3.3.2 Limitations of TASA at m18 and their solutions 76
3.3.2.1 *typedef* and dependent declarations 76
3.3.2.2 Vendor-Specific/Non-ANSI Extensions 76
3.3.2.3 preTASA 77
3.3.3 Evaluation 77
3.3.4 Cache line Randomisation 78
3.3.5 Memory Overheads 79
3.3.6 Average Performance 81
3.3.7 pWCET Estimations 82

4 Conclusion **83**

Acronyms and Abbreviations **84**

References **85**

Executive Summary

This deliverable reports on the research-oriented part of the work performed in the task T2.6 as well as the refinement of the tasks T2.3 and T2.4 as reported in D2.5, D2.6 and D2.7 at m18.
In particular, this document presents the updates achieved in the last reporting period in the research-oriented work carried out in WP2, for rendering time-composable the research-oriented RTOSes considered in PROXIMA, and for providing software-only solutions to timing randomisation for the COTS processors targeted in the project. The material presented in document includes the evaluation of all proposed solutions on the multi-core processors of interest.
The research-oriented software stacks consists of three distinct RTOSes, one for each COTS platform (FPGA, P4080, AURIX) and two Software Randomisation tool chains, one dynamic variant (i.e., with randomization performed at run time) for the FPGA and the P4080, and one static variant (i.e., with randomization performed at compile time on a per executable basis) addressing the specific needs of the AURIX processor and of its application domain.
The combination of the elements of this software stack on each of the target platforms provides the desired properties required to perform MBPTA analysis on COTS platforms namely timing composability and time-randomisation.
This document is organized as follows: Section 1 provides a general introduction to this line of action in the wider context of the project; Section 2 presents the implementation and evaluation work performed on the research-oriented RTOSes; Section 3 illustrates all software randomisation tool chains for all processor targets, along with the corresponding evaluation results, obtained with synthetic kernels (programs) or benchmarks. The use of these technologies in the industrial case studies is presented in the WP4 reports due at M36, in particular D4.8. Section 4 concludes the document.

1 Introduction

In order to address the complexity of modern architectures, the PROXIMA approach has been centred around capturing, understanding and taming *sources of jitter (SoJ)*.
In the base case of a single task executing on a single-core architecture without an operating system, the only identified SoJ arises from the memory layout of the software and its interaction with cache memory hierarchies. Previous studies have shown that the **memory layout** impacts significantly the execution time [12] even in the presence of a single cache used for instructions [13]. In order to control this SoJ, we employ *software randomisation* means in order to randomise the memory layout of the task. This further provides a time-randomised behaviour in the task's execution time, with independent and identically distributed (i.i.d.) characteristics, which enables the use of MBPTA. This enables arguing on the jitter caused by the cache in probabilistic terms, giving each potential cache layout a probability of appearance [8].
In more complex and more realistic scenarios involving a *real-time operating system (RTOS)*, the operating system itself adds other SoJ to the problem domain. When the task calls an RTOS service, the RTOS activity alters the hardware state and especially the cache contents, that may cause a severe impact on the task timing after the service has been completed. This complicates the timing analysis process since it requires that the application has to be analysed together with the RTOS.
As a solution to this problem, we use an RTOS design which provides *constant-time* services, thus enabling *timing composability*. This means that the application and the RTOS can be analysed in isolation and each component represents a simple additive factor in the total execution time.
Moving towards a single-core system with multiple tasks, other SoJ stem from the interactions between the tasks, which time-share the processor resources. The most evident example of that kind is **preemption**, which we address by modifying the RTOS scheduling to provide run-to-completion semantics.
Finally, in a system with multiple tasks and multiple CPUs, there are additional SoJs. Some arise from **task migration** that introduces various types of delays (at the ends of the migration and, potentially for all cores owing to cache pollution effects in memory) and considerably complicates timing analysis. This issue is addressed by modifications in the scheduling and adoption of partitioned solutions, which favours time composability. Further SoJ emerge from **hardware resource sharing** between tasks executing in different processors, such as shared buses and caches. For resources that provide hardware features for segregation such as partitioning in shared caches, we employ them in order to create per core partitions to enable time composability. On the other hand, for resources without such features as shared buses, we rely on the time-randomised behaviour provided by software randomisation methods in order to probabilistically characterise the worst-case behaviour of each task (see D3.8).
Overall, when all these solutions are combined together, they enable the use of MBPTA on the multicore COTS platforms used in PROXIMA and allow yielding time-composable estimates, while they further help keeping the whole analysis process simple.

In this deliverable report we provide updates on the employed solutions designed to address the SoJ summarised above in the last reporting period of the project, and their experimental evaluation.
In the presentation of this material, we follow a two-level organisation: Chapter 2 describes the RTOS features devised in the project to obtain time-composable execution behaviour in the RTOS, for the research-oriented RTOSes considered in PROXIMA, and presents the experimental evaluation of the goodness of fit of the proposed modifications; Chapter 3 focuses on software randomisation solutions, their modifications and their experimental evaluation. Each chapter is further subdivided per target platform to address the specific requirements in each case.

2 Research-oriented RTOSes

PROXIMA set itself the ambition of pursuing its overall goals across two parallel lines of software development action: one centred around industrial-quality technology, which warranted faster exploitation paths; the other centred research-type technology prototypes, which were meant to explore more innovative solutions with more agility, risking less resistance friction than with industrial-quality product baselines.
This ambitious plan rested on two critical assumptions:

- sufficient availability of the processor boards to both development groups (the industrial-ready one and its research-type parallel) for the port and the refinement of their respective technology.

- sufficient energy at the industrial users to port their use cases and repeat the MBPTA experiments on both technology solutions.

As the project had a clear, undisputed slant on industrial exploitation, precedence was systematically given to the industrial-quality technology development, both in access to the processor boards and in interacting with the industrial users in WP4. This arrangement caused two distinct contingencies.

1. The exposure of the research-type prototypes to the industrial use cases was limited to the automotive use case, thanks to the fact of it having no industrial-ready RTOS solution to consider.

2. The P4080 processor boards quickly became a scarce resource, which – owing to the extraordinary complexity of internal architecture and its very steep learning curve – could not conceivably be physically time shared in short rounds, but had to reside at the designated user for long time spans. Remote access was also not a viable back-up route as porting, debugging and consolidating an RTOS on a processor board need numerous and very fast turn-around runs, which cannot be conceivably warranted with a limited-time remote-access resource.

As a result of the above contingencies:

- The development of the research-type RTOS for the P4080 (which was a task of UPD) was discontinued before its port could reach an implementation status apt for worthwhile experimental evaluation owing to insufficient availability of the processor board and the inability of the WP4 industrial member that was the intended user of it to devote energy for the port of an (avionics) application to it.

- The development of the research-type RTOS targeted for the FPGA COTS (also in charge to UPD) was successfully completed, but its evaluation was performed outside of PROXIMA, as the WP4 industrial member that was the intended user of it was unable to devote energy to adapting a (space) application to it. The proceedings of that line of action are discussed in Section 2.1.

For this reason, this report does not include material on the research-type RTOS destined for use on the P4080. The UPD effort detracted from the RTOS development for the P4080 was transferred to supporting the development and execution of the automotive use case on the AURIX processor, in addition to bringing to completion the time-composable adaptations of Erika Enterprise, the research-type RTOS selected to that end. The proceedings of that line of action are discussed in Section 2.2.

2.1 FPGA COTS

The research-oriented that was singled-out in PROXIMA for use with the FPGA COTS was the SMP (symmetric multiprocessor) version of RTEMS[1], owing to its industrial relevance to ADS and to the European Space Agency) in the space domain.

The expectation, for PROXIMA and for UPD, was that ADS would be able to replicate some of their use-case experiments using an RTEMS-SMP based execution stack and therefore provide feedback on the usefulness of the modifications developed to it by UPD in PROXIMA, with the intent to facilitate the use of the PROXIMA timing analysis solutions for the FPGA COTS target. Unfortunately, this proved not possible as ADS spent all of its available energy to complete their planned set of use-case experiments using the industrial RTOS option. Luckily however, the proceeds of the work done in WP2 on RTEMS-SMP are going to be reflected back in the RTEMS-SMP refinement project that is being conducted by the European Space Agency.

In this section we provide an update on the evaluation of the effectiveness of the modifications that UPD developed to the RTEMS-SMP code-base to improve the composability of its timing behaviour and therefore have very small, if not null, perturbation effects on the measurement observations performed by the user on the application programs subject of worst-case execution time analysis.

RTEMS was a big and complex operating system. Its SMP version is considerably bigger and more complex. Interestingly, in the process of developing the SMP adaptations of RTEMS a number of modifications to the way of working of single-CPU RTEMS had to be implemented, which culminated in the slow and complex process of removing the so-called "giant lock" that the old code-base used to protect access to internal data against reentrant code.

The process of rendering RTEMS-SMP fully time-composable was not sustainable within PROXIMA, which had the energy to focus only on a central subset of services. To prove the goodness of fit of our modifications (or re-implementations), we compare the timing behaviour of our modified version of RTEMS-SMP[2] against the original version of it.

To give more room to the presentation of the evaluation results, we omit discussing details of the implementation, whenever that omission does not negatively affect the reader's understanding.

RTEMS offers the end user a lot of freedom, which however often is a double-edged sword. To prevent misuse of kernel primitives and to ensure conformance with the intention of our changes, we describe how those primitives should be used to obtain sound and analysable applications.

Our changes to RTEMS have not been merged yet in the master branch of RTEMS at the time of this writing. We are pursuing that merge outside of PROXIMA. For the purposes of this report we describe how to obtain a working version of our modified RTEMS code-base.

Unless otherwise stated, our evaluation experiments have been performed in the

[1]RTEMS was originally designed for single-CPU processors and, until very recently, lacked a solid version fit for working on multi-core processors. UPD participated in the initiative the created the first SMP code-base for RTEMS, under funding from the European Space Agency.

[2]In the following we use the RTEMS acronym as a synonym for RTEMS-SMP.

LEON3 PROXIMA FPGA[3] *without* randomisation. We did indeed perform some initial experiments to evaluate the impact of hardware randomisation on our modifications for time composability; we could not complete that evaluation in the time-frame of PROXIMA, but we plan to continue it outside of the project. We chose however to not include those initial results in this report.
Unless otherwise stated, our experimental results are presented with comparative charts. All numerical values are reported in nanoseconds. The left part of each chart depicts the measurements from our modified version, while the right chart (with gray background) depicts the measurements obtained with the original version of RTEMS (dated May 10th, 2015 at commit be0366bb62ed4a804725a484ffd73242cd4f1d7b). Each chart provides four pieces of information: the minimum observed value (blue line), the maximum observed value (red line), the median (black thick line) and the region spanning from the first and third quartile (green region)[4]. All measurements have been taken with lightweight ad-hoc tracing support developed in house.

2.1.1 Task Manager

The task manager comprises all those directives related to the management of tasks. Table 1 reports the primitives that we considered in our effort, which may induce significant runtime overhead, and are related to scheduling decisions. Interestingly, the first four primitives in Table 1 should not be used during runtime, but only during the initialisation phase of the system. Yet, their operation has repercussions on the runtime behaviour.

Table 1: Selected primitives of the Task Manager

TASK_CREATE
TASK_IDENT
TASK_SELF
TASK_START
TASK_SUSPEND
TASK_RESUME
TASK_WAKE_AFTER

The runtime behaviour and overhead caused of such primitives is strictly determined by the scheduler used in the application. In order to cause those primitives to exhibit a time-composable behaviour, it is opportune to choose among the options supported by the RTEMS code-base a scheduling algorithm that can reproduce – in the SMP environment – the same execution conditions given in a single-CPU processor, which is partitioned fixed-priority scheduling (P-FP). In Listing 1 we show a possible configuration for an application that must run in a system with CPU_COUNT processors and where each partition has PRIO_COUNT priority levels.

Listing 1: Example configuration for a P-FP system

```
#define CONFIGURE_SMP_MAXIMUM_PROCESSORS CPU_COUNT
#define CONFIGURE_SCHEDULER_PRIORITY_SMP
```

[3]The LEON3 PROXIMA FPGA platform is a system-on-chip with four LEON3 SPARC V8 processors running at 80MHz, with 4x4KiB Data Cache and 4x4KiB Instruction Cache.

[4]This is where 50% of the samples (centred around the median) are located.

```
#include <rtems/scheduler.h>
RTEMS_SCHEDULER_CONTEXT_PRIORITY_SMP ( 0 , PRIO_COUNT ) ;
. . .
RTEMS_SCHEDULER_CONTEXT_PRIORITY_SMP (CPU_COUNT-1, PRIO_COUNT ) ;

#define CONFIGURE_SCHEDULER_CONTROLS \
RTEMS_SCHEDULER_CONTROL_PRIORITY_SMP ( 0 , 0 ) , \
. . .
RTEMS_SCHEDULER_CONTROL_PRIORITY_SMP (CPU_COUNT-1, CPU_COUNT-1)

#define CONFIGURE_SMP_SCHEDULER_ASSIGNMENTS \
RTEMS_SCHEDULER_ASSIGN ( 0 , RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY ) , \
. . .
RTEMS_SCHEDULER_ASSIGN (CPU_COUNT-1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY )
```

As that the task set is statically defined at design time, performing a dynamic search of a task while the system is running based on the alphanumeric name of the task is definitely not a sound solution. The identifier of each task is produced as side effect of the TASK_CREATE primitive. It is therefore opportune to save that identifier in a specific location so that every other task that needs it can find it directly without using the TASK_IDENT primitive.
In order to create a periodic task, the user should call the TASK_WAKE_WHEN primitive, as depicted in Listing 2. That primitive should be used to set the time of the next release of the current task, and the call to this primitive results in the task to suspend its execution.

Listing 2: Body of a periodic task

```
static void periodic_task(rtems_task_argument arg){
  while(true){
    //... do some work ...
    rtems_task_wake_when(...);
  }
}
```

The implementation of sporadic tasks should be based on the TASK_WAKE_WHEN primitive in order to guarantee the minimum inter-arrival time. As shown in Listing 3, after the wait there should be a barrier that blocks the execution of the task until some external trigger arrives. The usage of events for this purpose is further discussed later in this section.

Listing 3: Body of a sporadic task

```
static void sporadic_task(rtems_task_argument arg){
  while(true){
  //... do some work ...
  rtems_task_wake_when(...);
  rtems_event_receive_sporadic_activation(...);
  }
}
```

2.1.2 Evaluation

We chose to not directly evaluate the primitives, but to concentrate instead on the internal operations of the scheduler:

- yield (yielding a task)
- block (blocking a task)
- unblock (unblocking a task)
- priority prepend (changing the priority of a task)
- priority append (changing the priority of a task).

In all cases, we varied the number of tasks in the system (the variation is shown on the X axis of the chart in Figure 1) and observed that the macroscopic behaviour of those internal operations did not change after our modifications. In fact, the original implementation of the scheduler was sufficiently time composable of its own. The good news however, was that the modifications we applied to enable the implementation of novel, SMP-capable, synchronisation problem for sharing global resources across tasks pinned to cores (which we discus separately later in this report) did not negatively affect the good properties of the scheduler.

2.1.3 Semaphore Manager

The semaphore manager comprises all those directives related to the management of semaphores. Table 2 reports the primitives that we considered in PROXIMA.

Table 2: The semaphore manager operations

SEMAPHORE_CREATE
SEMAPHORE_IDENT
SEMAPHORE_OBTAIN
SEMAPHORE_RELEASE
SEMAPHORE_FLUSH

We focused only on those primitives that can actually produce runtime overhead. In this regard, the first two primitives of Table 2 should not be used during run time, but only at system initialisation. Our focus on this group of primitives was entirely centred on the implementation of the SMP-capable semaphore support known as MrsP [4].
RTEMS exposes several types of semaphores through the same interface. A specific type of semaphore is selected based on the input parameters that the user supplies to the SEMAPHORE_CREATE primitive. Listing 4 shows the parameters to use to issue the creation of an MrsP semaphore.

Listing 4: Creation of an MrsP Semaphore

```
rtems_semaphore_create(
  name_of_the_semaphore,
  1, /*mustbeone*/
  RTEMS_MULTIPROCESSOR_RESOURCE_SHARING|RTEMS_BINARY_SEMAPHORE,
  ceiling_priority, /* initial and temporary ceiling priority for all
  partitions. Should be updated locally with
    rtems_semaphore_set_priority */
  &resource_id);
```

An MrsP semaphore is a way to synchronise several tasks such to consistently use some shared object in mutual exclusion. If such tasks reside in the same partition, then the behaviour of MrsP is the same as the well-known ceiling semaphore SRP (uniprocessor Stack Resource Protocol), in its adaptation for single-CPU fixed-priority scheduling systems [3]. If such tasks reside in different partitions, then MrsP provides a helping mechanism that is crucial to reduce the wait time of tasks pending on the semaphore.
Obtaining and releasing an MrsP semaphore does not come for free. In case the critical section is very short, it may be more convenient to make critical sections non-preemptive. In an SMP environment, this means disabling dispatching in the specific partition and then acquiring a spin-lock (to prevent the parallel execution of the critical section from a task in a different partition). In RTEMS, this can be

achieved by enclosing the critical section between _Thread_Disable_dispatch and _Thread_Enable_disaptch primitives. Notably, however, the spinlock acquired in that way is the notorious "giant lock" that protects most of the kernel structures of RTEMS (from reentrant calls in the single-CPU version and from parallel calls in the initial SMP version. At the time of this writing, an ESA-funded activity is currently removing the giant lock from the SMP codebase). It therefore follows that the use of such primitives can lead to delays in the execution of the kernel primitives in any partition, which is a rather bad perturbation.
We reckon that for critical sections longer than 30 μs it is advisable to use our MrsP semaphores. Indeed, this maximum duration that we have observed that is spent inside the kernel in non-preemptive mode while trying to acquire or release an MrsP resource. After obtaining the resource, the task remains preemptive (while running at ceiling priority), meaning that all other tasks in the same partition can be unconditionally delayed at most for the time spent by that tasks while being non-preemptive inside the SEMAPHORE_OBTAIN primitive. Using non-preemptive critical sections longer than 30 μs means that all the tasks in the same partition will unconditionally suffer for the whole execution of the critical section.
For MrsP semaphores, the SEMAPHORE_FLUSH operation is not defined, hence not used.
At the time of this writing, there are the following limitations with the time-composable version of MrsP:

- an application can use at most 32 MrsP semaphores. This is a temporary limit since the time-composable version is based on bit-masks of uint32_t (therefore 32 bits). This means that the macro CONFIGURE_MAXIMUM_MRSP_SEMAPHORES cannot be greater than 32.

- some scheduling primitives have been changed in order to fit our version of MrsP. Not all the primitives of all schedulers have been changed. The only scheduler known to work for sure with our implementation of MrsP is the Deterministic Priority SMP Scheduler.

- our version of MrsP does not support timeouts (this has been an intentional choice).

- our version of MrsP forbids the use of blocking operations. For reference, the most common blocking operations are:

    - rtems_rate_monotonic_period
    - rtems_task_wake_after
    - rtems_task_wake_when
    - rtems_task_suspend
    - rtems_task_set_scheduler
    - rtems_event_receive

2.1.4 Evaluation

We used five tests to perform comparative measurements on the following primitives:

- Obtain: the body of the lock procedure, until either the resource is locked (when the resource is free) or the task start spinning (when the resource is already occupied). In the latter case, we omit our results for the time spent inside the kernel from the time a task stop spinning to its actual exit from the SEMAPHORE_OBTAIN primitive since there are virtually no operations to perform.

- Release: the body of the unlock procedure.

- Help: the body of the helping protocol, called whenever a task holding a resource is preempted. This primitive must understand if and where are the tasks that can help the preempted resource holder.

Test 1 (see Figure 2) Goal: to determine the cost to obtain and release a resource while all pending tasks are active. The x-axis variation represents how many tasks are pending at the time of the obtain and release operation.
When the resource is free (x = 0), there are less operations to perform in order to obtain it. Otherwise, the time is mostly constant.
Our version of the protocol has a nearly-constant time behaviour, while the original version is linearly dependent on the number of tasks pending on the resource.

Test 2 (see Figure 3) Goal: to determine the cost to obtain an already occupied resource while its owner is not executing. The X-axis variation in Figure 3 represents the number of previous pending requests, whose tasks are not available to offer help (i.e., tasks that have already requested the resource and started spinning, but have been preempted since).
The chart shows no measurements for X=0, as this point in time captures the first task to request the resource, when the resource is free, which obviously obtains it at no cost. The behaviour we observe is similar to the one seen for Test 1, except that in this case we have to make a scheduling decision to launch the helping protocol when needed.

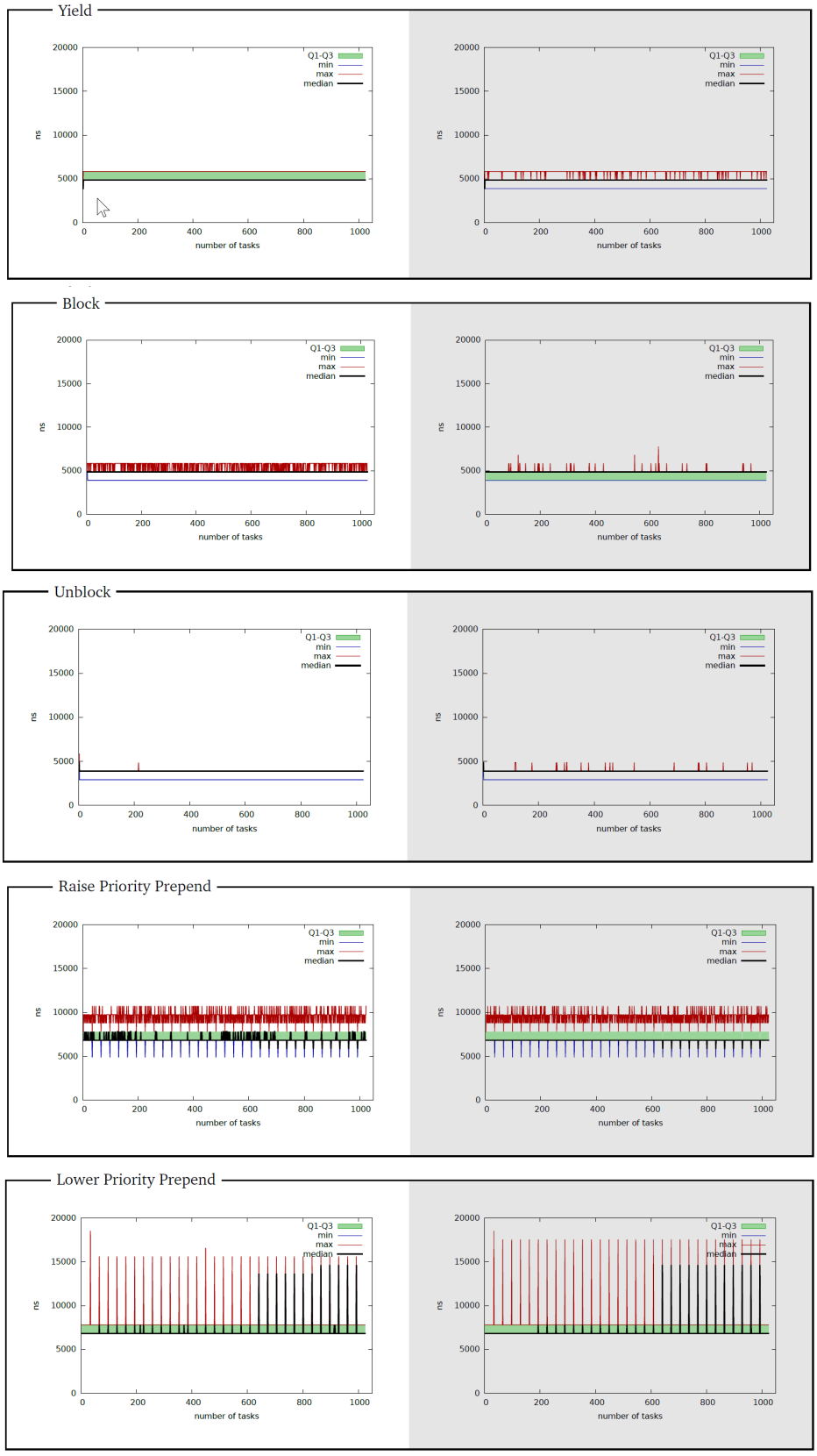

Figure 1: Comparative evaluation for Task Manager primitives after our modifications

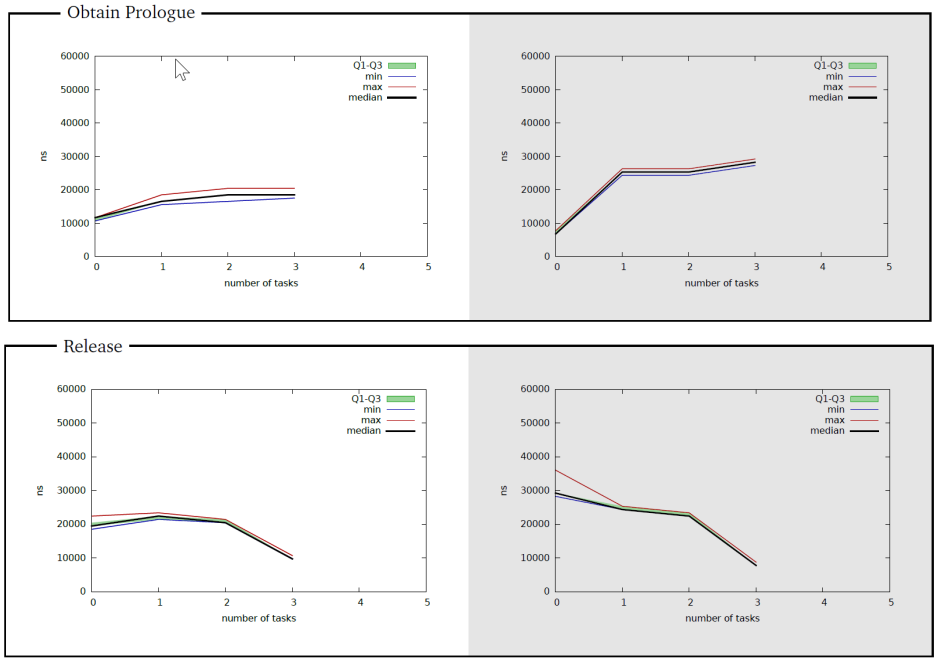

Figure 2: Comparative evaluation for the Semaphore Manager primitives after our modifications: Test 1

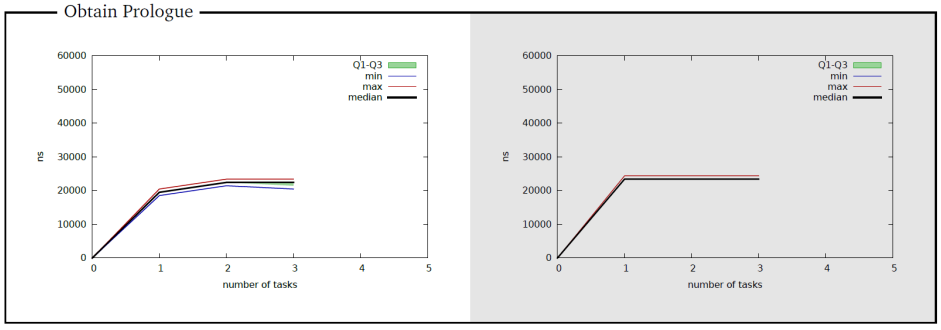

Figure 3: Comparative evaluation for the Semaphore Manager primitives after our modifications: Test 2

Test 3 (see Figure 4) Goal: to determine the cost to release a resource and make the next-in-line pending task the new resource holder. All tasks are pending. The X-axis variation represents the number of pending tasks that are active (i.e., that are spinning). Specifically, the active tasks are counted backward, that is, if there are 2 active task, they are the last 2 tasks that have requested the resource. Therefore, while upgrading the first pending task to the role of resource owner, the release procedure must find the processor (if any) in which make the new holder execute. Samples are generated from task 0.

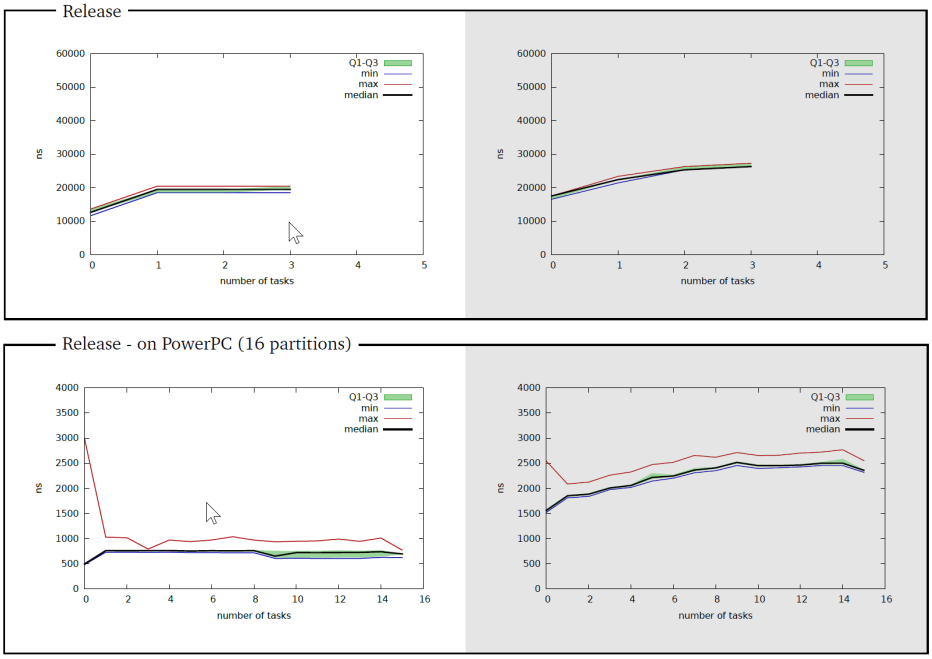

Figure 4: Comparative evaluation for the Semaphore Manager primitives after our modifications: Test 3

Our version of the protocol is mostly constant, while the original version is linearly dependent to the number of tasks pending on the resource. This is best seen on a platform where a higher degree of parallelism is available. The next chart represents the evaluation performed on a PowerPC with 16 cores.

Test 4 (see Figure 5) Goal: to determine the cost to select the task that must help the resource holder when it preempted. All tasks are pending.
The X-axis variation represents the number of pending tasks that are active (i.e., that are spinning). Specifically, the active tasks are counted backward, that is, if there are 2 active tasks, they are the last 2 tasks that have requested the resource. The chart shows no measurements for x = 0, as that point in time represents a lock-holder task that cannot help itself because it is alone in the waiting queue.
Our version of the protocol exhibits a nearly-constant timing behaviour, while the original version is linearly dependent on the position of the spinning task that can offer help inside the FIFO queue of the MrsP resource. This improvement effect is best seen on a platform with a higher degree of parallelism than PROXIMA's FPGA

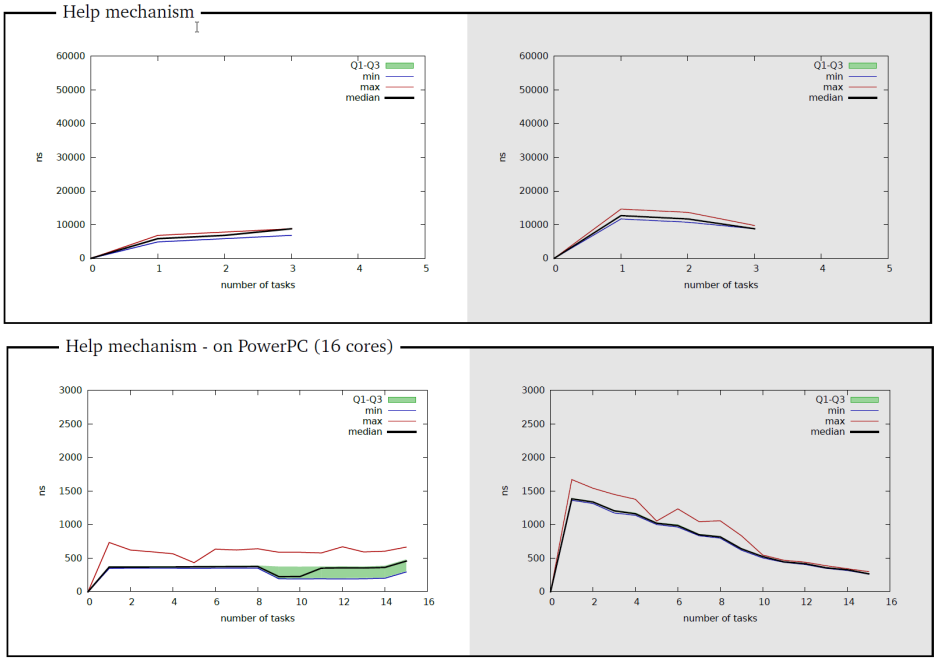

Figure 5: Comparative evaluation for the Semaphore Manager primitives after our modifications: Test 4

COTS. We show that in the lower chart in Figure 5, which plots the evaluation results obtained on a PowerPC with 16 cores.

Test 5 (see Figure 6) Goal: to determine the cost to obtain and release a resource when tasks are transitively nesting resources.
Specifically, there are 2 tasks nesting resources (on CPU 0 and on CPU 3). These two tasks nest exactly 2 resources each. The other tasks (if any) only request one resource that is already being locked by the nesting task. Eventually, a chain of 32 nested resources forms, where the first task requesting resource 0 (at point X=0 in the chart) must wait for the last task that requested resource 31 (x=31).
Both versions do not have a time-constant behaviour. Our version is better, as it linearly depends on the depth of nesting, whereas the old version linearly depends on both the depth of nesting and the number of tasks in the wait queue (thus having two orthogonal traversals).
Interestingly, these considerations hold for both primitives. Furthermore, in the old version, not only the tasks that have nested resources pay the price of a high overhead, but also the tasks do not take any part in resource nesting pay a high toll too. This is in accord with what we saw in Test 3: the release of an MrsP resource in the original version of the protocol is linear dependent to the number of tasks that wait (directly or indirectly through nesting) on the same resource.

2.1.5 Event Manager

The event manager comprises all those directives related to the signal passing. Table 3 reports the RTEMS original primitives we considered initially.
In the original semantic of RTEMS, a task can wait simultaneously for several

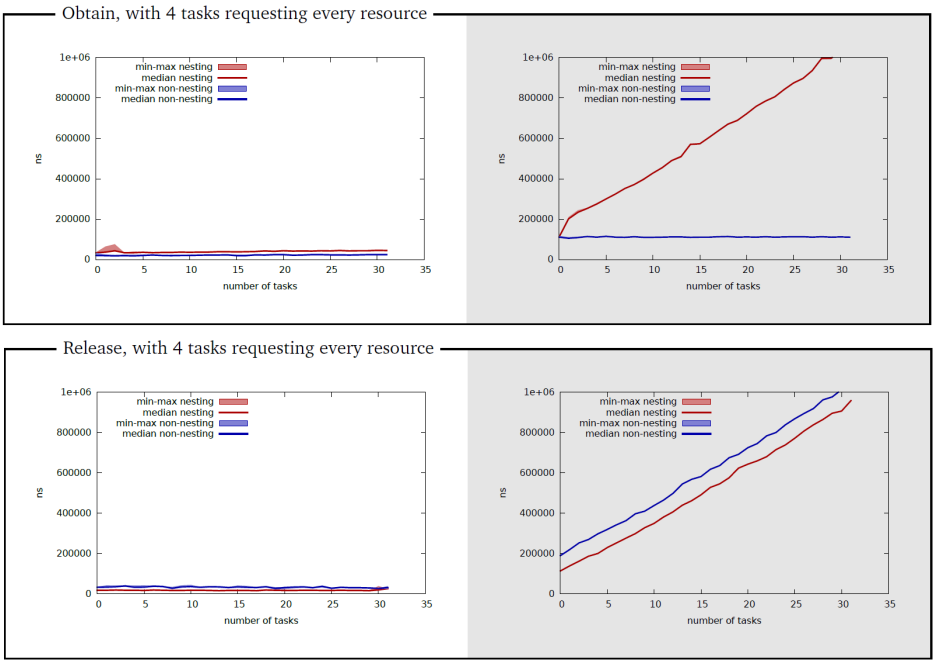

Figure 6: Comparative evaluation for the Semaphore Manager primitives after our modifications: Test 5

Table 3: Notable primitives of Event Manager

EVENT_RECEIVE EVENT_SEND

events or a task can wait for one specific event. While this approach can support a variety of scenarios (e.g., simulate complex barriers based on AND/OR conditions), such a large extent of expressivity can cause problems for time-predictable real-time applications. We therefore largely streamlined the supported semantics: events are used solely to prompt the release of a sporadic task (see Listing 3 for the skeleton of a sporadic task envisioned within our semantics) or to synchronise two tasks on a simple event/notification/signal (e.g., wait for an external command, wait for the completion of an action). Other kinds of synchronisation must be achieved with other means (e.g., semaphores).
Interestingly, events are used a lot inside the kernel of RTEMS: we did not take the risk to directly change them in order to conform them to our stricter semantics. Instead, we chose to provide a novel interface to send or receive events, which would be free from intersections with the old RTEMS events. Table 4 reports the interface of the event manager that we defined.

Table 4: Revisited set of Event Manager primitives

EVENT_RECEIVE_LOCAL
EVENT_SEND_LOCAL
EVENT_RECEIVE_SPORADIC_ACTIVATION
EVENT_SEND_SPORADIC_ACTIVATION

As a first step, we differentiate between events sent to tasks residing in the same

partition (per-core events) and events sent to tasks residing in a remote partition (extra-core events). We do this because, in our scenario, when a task sends an event, it means that it is completing its execution: in our scenario an event is sent only as the last action of a thread, to notify that its work is complete and to resume the work of someone else, in the form of a classic work pipeline.
If the event is a per-core event, this signal will put a task of the same partition in the ready state: since the sending task is going to complete, it can directly manage the scheduling decision for the receiving task since such overhead must be paid in any case by the partition. However, when a task sends an extra-core event, we want to avoid the senders partition to pay for the overhead of the scheduling decision on the remote partition. In this case, we want the event to be asynchronous: the overhead produced by the scheduling decision must be paid by the remote partition. The only permitted extra-core events are those that cause the release of a sporadic task.
The evaluation and implementation of our customised events is still ongoing. With the supplied patch of RTEMS, we implemented support for per-core events:

- EVENT_SEND_LOCAL
- EVENT_RECEIVE_LOCAL

Listing 5 shows the interface that we provide for such events (this interface is identical to the one in the original RTEMS manager).

Listing 5: Per-core events API

```
rtems_status_code rtems_event_send_local(
  rtems_id  id,
  rtems_event_set event_in
);

rtems_status_code rtems_event_receive_local(
  rtems_event_set event_in,
  rtems_option  option_set,
  rtems_interval  ticks,
  rtems_event_set *event_out
);
```

At the time of this writing, the extra-core events are still not implemented inside the kernel, but a proof-of-concept implementation have been developed and trialled successfully. Listing 6 shows the interface we envisage for those events.

Listing 6: Extra-core events API

```
rtems_status_code rtems_event_send_sporadic_activation(
  rtems_sporadic_event  event
);

rtems_status_code rtems_event_receive_sporadic_activation(
  rtems_sporadic_event  event
);
```

2.1.6 Evaluation (limited to per-core events)

In this group of evaluation experiments we only considered the original SMP-version of the RTEMS because its codebase was good enough to only require very minor adaptations for our project, with virtually null effect on the execution-time behaviour. The X-axis variation in the charts represents the size of the event set: through a single primitive it is possible to send/receive simultaneously several events. Events set are built with bitwise-or of bit-masks.

Test 1 (see Figure 7) Goal: to determine the cost to send an event to a thread that is not waiting for any event.

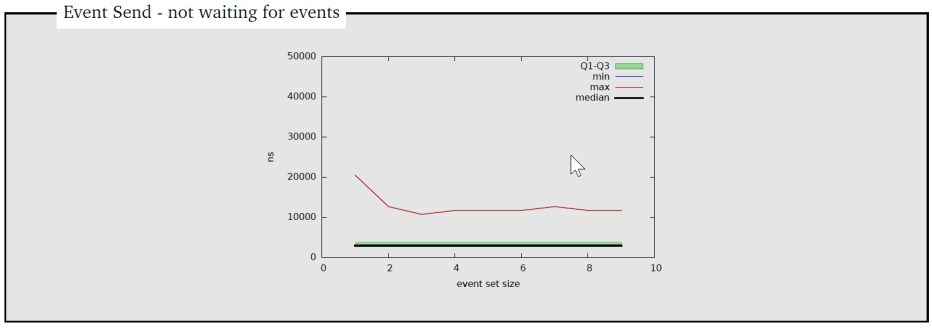

Figure 7: Comparative evaluation for the Event Manager primitives after our modifications: Test 1

What we observe in this case is essentially constant-time behaviour.

Test 2 (see Figure 8) Goal: to determine the cost to send an event to a thread that is waiting for an event, but without satisfying the condition on which the receiver is pending.

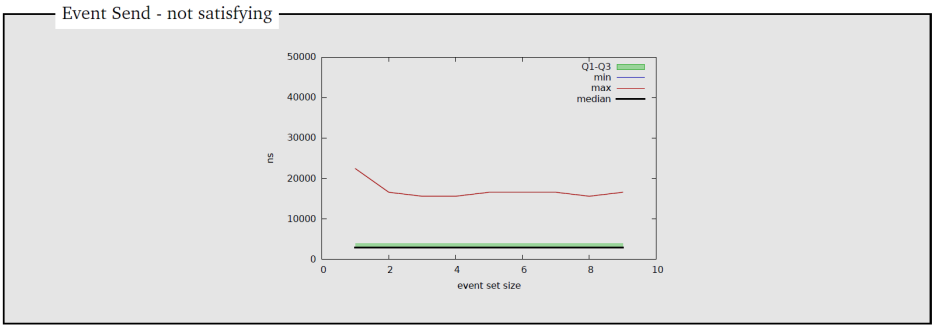

Figure 8: Comparative evaluation for the Event Manager primitives after our modifications: Test 2

Also in this case we see constant-time behaviour.

Test 3 (see Figure 9) Goal: to determine the cost to send an event to a thread that is waiting for an event, and satisfying the condition on which the receiver is pending.
Again, we see constant-time behaviour, but also greater overhead because the kernel has to unblock the receiver.

Test 4 (see Figure 10) Goal: to determine the cost to read the events pending on a task.
Not surprisingly, given the nature of the primitive, here we see perfect constant-time behaviour.

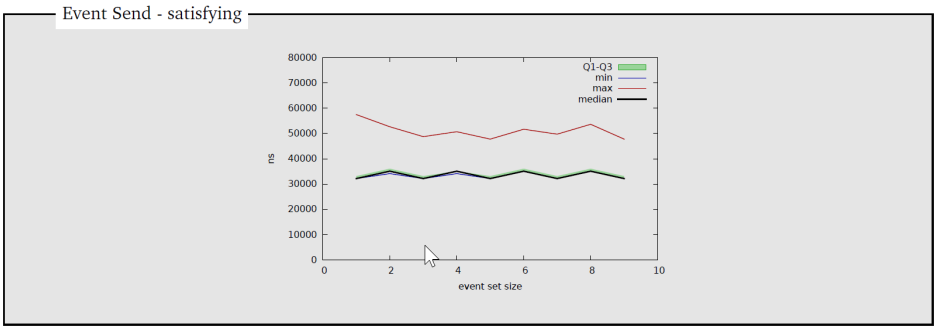

Figure 9: Comparative evaluation for the Event Manager primitives after our modifications: Test 3

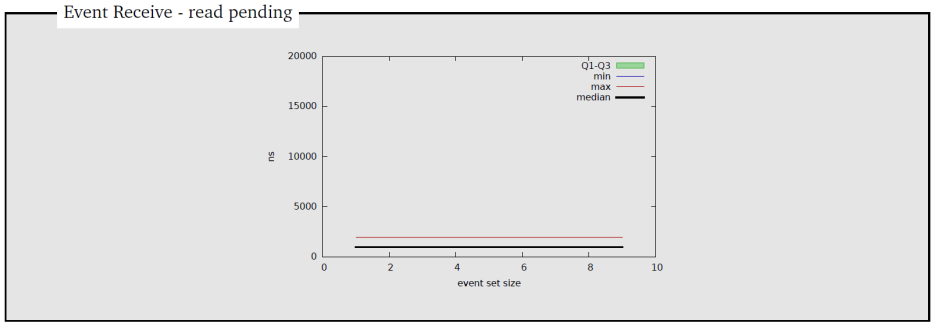

Figure 10: Comparative evaluation for the Event Manager primitives after our modifications: Test 4

Test 5 (see Figure 11) Goal: to determine the cost to receive an event that is already pending on the task.
Again, constant-time behaviour.

Test 6 (see Figure 12) Goal: to determine the cost to receive an event that is not already pending.
Symmetrically to Test 3, here we see constant-time behaviour, but also greater overhead because the kernel has to block the receiver.

2.1.7 Utilities

Toolchain The toolchain used to compile RTEMS (and our patch) is based on the rtems source builder (RSB). It is useful to note that the specific version of RTEMS relates to a specific version of the toolchain. In case the toolchain gets updated, it is necessary to work with the specific toolchain dating the same period of the specific commit of RTEMS in order to avoid problems. The specific toolchain can be obtained following the steps shown in Listing 7.

Listing 7: Obtaining the toolchain

```
$git clone git://git.rtems.org/rtems-source-builder.git
$cd rtems-source-builder
$git checkout b65c131f2e11e352fde6efa0ec2fe5000dad3a4a
```

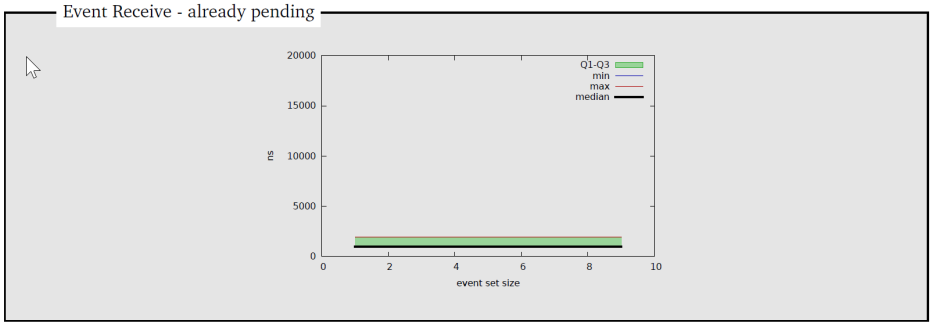

Figure 11: Comparative evaluation for the Event Manager primitives after our modifications: Test 5

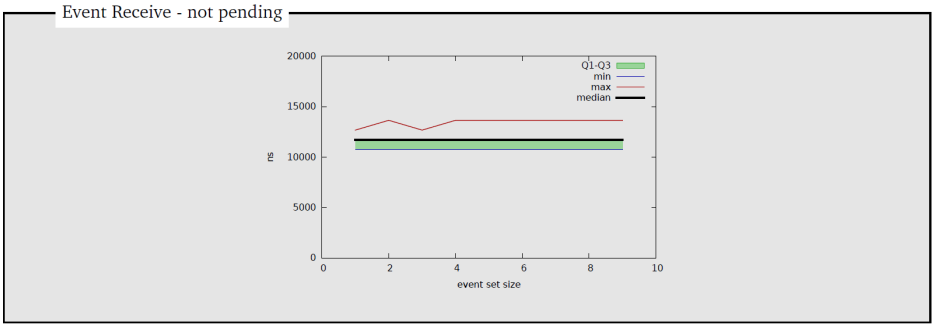

Figure 12: Comparative evaluation for the Event Manager primitives after our modifications: Test 3

To compile the toolchain, please refer to `https://devel.rtems.org/wiki/Developer/Tools/RSB`.

RTEMS and Patch The supplied patch works against a specific (old) commit of RTEMS: 40d24d54ab59fdb2e4133128bf184ec8935f3545. Following the steps of Listing 8 to correctly apply the patch. There could be some warnings about trailing whitespaces: this is not an error.

Listing 8: Getting and patching RTEMS

```
$git clone git://git.rtems.org/rtems.git
$cd rtems
$git checkout -b timecomposable 40d24d54ab59fdb2e4133128bf184ec8935f3545
$git apply --check $PATH_TO_THE_PATCH/timecomposable.patch
$git am --signoff < $PATH_TO_THE_PATCH/timecomposable.patch
```

To avoid frustrations, we suggest following the steps specified below to compile RTEMS:

1. run the bootstrap inside the rtems folder;

```
cd $RTEMS_PATH
./bootstrap -g
```

2. export the path of the toolchain;

3. create a folder outside the source tree of RTEMS;
4. in that folder, run the following command:

```
$PATH_TO_YOUR_RTEMS_SOURCE_FOLDER/configure \
prefix=$PATH_TO_THE_TOOLCHAIN/toolchain \
target=sparc-rtems4.11 disable-itron enable-smp disable-cxx \
disable-docs disable-tests disable-posix CONSOLE_USE_INTERRUPTS=0 \
enable-rtemsbsp=leon3
```

5. run `make -j<N>` where `<N>` is the number of host processors you want to use during the compilation phase to speed up the process.

The repo includes the following three files:

1. timecomposable.patch: the patch that needs to be applied to RTEMS.
2. eventSporadic.c: an RTEMS application, showing the proof-of-concept of asynchronous events.
3. compileFile.sh: a bash script that can be used to compile an RTEMS application comprised of one single file. If the application has more files, then it is necessary to customise it. In all cases, it is necessary to update the path variables inside the script to point to the correct locations.

2.2 AURIX

The research-oriented RTOS that we selected for the AURIX was Erika Enterprise, a free-of-charge, open-source, certified OSEK/VDX-compliant RTOS distributed by Evidence (`http://www.evidence.eu.com/`), which targets various single-core and multi-core micro-controllers, including Infineon's Aurix Tricore by Infineon. At the time of this writing, the Erika Enterprise code-base also covers most of the Autosar 4 requirements concerning multi-core support at RTOS level.
In this section we report on the work we performed to make Erika timing-composable and to support the automated application and evaluation of TASA (the static software randomisation variant specifically developed for AURIX and is described later in Section 3.3) as part of the build automation framework. The time-composable version of Erika (nicked Erika-TC) and its build-chain automaton were instrumental to enabling the execution of the automotive use case in PROXIMA. It important to note that, in spite of the magnitude and complexity of the modifications to be applied to the build automation to serve the needs of the PROXIMA analysis, Erika-TC proved an excellent enabler to measurement observations, in the way of providing no disturbance for state, contention and timing jitter to the execution at the application level.
An overview of the automated build infrastructure that we developed for PROXIMA including the time-composable version of Erika, the adaptations of its configuration utilities, and a number of artifacts that help the user run examples and evaluate various aspects of the execution-time behaviour of application programs running on the Aurix and Erika-TC is presented in [5].

2.2.1 System Configurations

Erika Enterprise is a highly flexible RTOS that supports multiple system configurations. Understanding how system configurations are defined and deployed is therefore one key aspect for the user. In Erika, system configurations are passed in input to an automation build framework.
As required by the OSEK standard, system configurations are plain-text specification that conform with the OSEK Implementation Language (OIL). The system specification passed to the automation framework is then translated into a set of configured source files that are compiled and linked together with the RTOS library and applications to produce the final executable. Of that material, only the functional specification (i.e., the task bodies) needs to be provided by the user.
One or more ELF files can be produced by the build process, depending on the chosen configuration and the target architecture. For the AURIX, Erika assumes a fully-partitioned configuration, with one executable image per active core. Erikas build process is summarised in Figure 13.
The OIL file typically consists of a set of objects (i.e., tasks, alarms, events, software resources, interrupt service routines, system hooks, etc.) and object attributes. In the following we describe the basic structure of an OIL file in single and multi-core configuration, which uses only the OSEK-compliant part of Erika and targets the Aurix Tricore. We use the terms application and task interchangeably.
In a single-core configuration, a single ELF is generated and deployed to CPU0. At the top level of the specification, we find a CPU object, which is the container of all other objects declared in the OIL. One single level below in the hierarchical

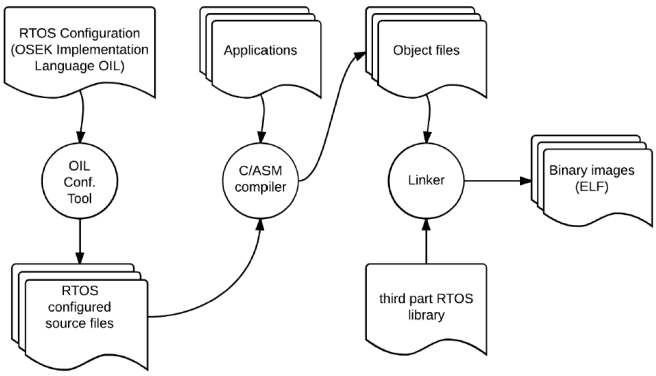

Figure 13: Erikas build process

configuration we find the OS block, which selects and configures all RTOS related features. Listing 9 shows this part of the OIL file.

Listing 9: Single-core OIL configuration

```
CPU test_application{
OS EE{
    MCU_DATA=TRICORE{MODEL=TC27x;
    };
    CPU_DATA=TRICORE{CPU_CLOCK=100.0;APP_SRC="code.c";COMPILER_TYPE=GNU;MULTI_STACK=TRUE;
    };
    KERNEL_TYPE=ECC1;
};
```

The CPU DATA object allows specifying core-specific parameters. The most important parameter is the file containing the functional application code (APP SRC). Listing 10 shows a simple example of application code, which – in this case – does nothing, as no application has been specified in the OIL, but it is sufficient to compile and cause the system to boot. More than one file can be included in the system configuration, but a single main function must be declared. The StartOs call never returns.

Listing 10: code.c: application code in single-core configuration

```
#include "ee.h"
  StartOS(OSDEFAULTAPPMODE);
  return 0;
}
```

Other useful parameters for the CPU DATA are: frequency of the core (CPU FREQ); compiler specific settings (COMPILER TYPE=GNU for the Hitech Tricore toolchain); the system stack configuration (the application stack can be either shared or private if the application needs its own private stack).
The OSEK standard defines four conformance classes to provide a fine-grained kernel configuration for kernel APIs and memory footprint. The KERNEL TYPE parameter specifies at which conformance class the kernel will be compiled, with values: BCC1, ECC1, BCC2, ECC2. The BCC1 and BCC2 classes only support basic applications that do not use synchronisation primitives (hence cannot enter a wait state) and do share the stack, therefore have a small RAM footprint. The ECC1 and ECC2 classs instead are for extended applications, which use synchronisation primitives and have their own private stack. In the latter case, some overhead is paid by the application owing to need to swap private stacks. Basic and extended applications can coexist in ECCx configurations; xCC2 classes allow

multiple application instances to be ready for execution at the same time, whereas xCC1 classes do not.
The multi-core configuration is similar to the single-core one, except that one full Erika instance is deployed onto each core. By default an ELF for each core is generated, even though a single-ELF configuration could be produced. Cores are activated according to the order specified in the Tricore specification manual. Adding multiple CPU DATA objects is sufficient to enable multi-core support. Different CPU DATA objects must be coherent on all parameters, except for the APP SRC one which includes code of applications assigned to the specific core. An ID discriminates objects associated to different cores and a master core must be elected specifying the MASTER CPU parameter.
Listing 11 shows an example using two cores (CPU0 and CPU1).

Listing 11: Multi-core OIL configuration

```
CPU test_application{
  OS EE{
    MASTER_CPU = "CPU0";
    MCU_DATA = TRICORE{
      MODEL=TC27x;
    };
  CPU_DATA = TRICORE{
    ID = "CPU0";
    CPU_CLOCK = 100.0;
    APP_SRC = "ee_tc27x_tracing.c";
    APP_SRC = "code.c";
    COMPILER_TYPE = GNU;
    };
  CPU_DATA=TRICORE{
    ID = "CPU1";
    CPU_CLOCK=100.0;
    APP_SRC = "ee_tc27x_tracing.c";
    APP_SRC = "code1.c";
    COMPILER_TYPE = GNU;
    };
  KERNEL_TYPE = BCC1;
  };
};
```

For the multi-core configuration, small changes are also needed in the application code: the master must activate other cores before initialising the OS as shown in Listing 12.

Listing 12: code.c: Application code in multi-core configuration

```
# include "ee.h"
int main(void)
{
  StatusType status;
  StartCore(OS_CORE_ID_1,&status);
  StartOS(OSDEFAULTAPPMODE);
  return 0;
}
```

2.2.2 Cyclic Tasks

Cyclic tasks issue jobs regularly over a fixed time period. In Erika, the periodic behaviour of cyclic tasks is obtained by attaching the release to a recurrent alarm. Listing 13 provides an example of periodic task. The TASK and ALARM objects are declared on the CPU object. Besides the name and the id associated to the cpu (CPU ID), a task is characterised by: (i) a priority value (PRIORITY) as the scheduling policy assumes fixed priorities; (ii) an (AUTOSTART) value which specifies whether the task is automatically executed after the OS initialisation phase; (iii) a SCHEDULE value which makes the task fully preemptible or non preemptible; (iv) a STACK value which specify whether stack is private or shared. A private stack is required when the task can be waiting for an event or a logic resource, and the kernel must be in ECCX configuration.

Listing 13: Configuring a cyclic task

```
TASK Task1{
  CPU_ID = "CPU0";
  PRIORITY = 1;
  AUTOSTART = FALSE;
  SCHEDULE = NONE;
  STACK = SHARED;
  };
ALARM Alarm_TASK1{
  COUNTER = system_timer;
  ACTION = ACTIVATETASK{
    TASK = Task1;
    };
  AUTOSTART = TRUE{
    ALARMTIME = 1000;
    CYCLETIME = 500;
    APPMODE = OSDEFAULTAPPMODE;
  };
};
```

The task activation is handled by the alarm object. Each periodic task must be associated to its own alarm object. The alarm is associated to a counter object that counts the hardware ticks from the system timer (residing in the STM peripheral). It is worth noting that the Tricore specification allows the STM peripheral to support two independent interrupt sources. The alarm starts counting immediately after system boot, and it is fired for the first time after the time specified by the `ALARMTIME` parameter. The period, instead, is specified by the `CYCLETIME` parameter.
Listing 14 shows the functional code associated to a cyclic task. Each such task must end with a `TerminateTak` call to release the CPU.

Listing 14: code.c: Cyclic task functional code

```
#include "ee.h"
volatile EE_UINT32 counter1 = 0;
TASK(Task1){
  counter1 ++;
  TerminateTask();
}
int main(void){
  StatusType status;
  StartCore(OS_CORE_ID_1, &status);
  StartOS(OSDEFAULTAPPMODE);
  return 0;
}
```

Task activation can also be chained to other tasks using the `ActivateChainTask` primitive. Tasks that neither activated by an alarm nor released by other tasks via a software signal (see below) are one-shot tasks.

2.2.3 Sporadic Tasks

Sporadic tasks are recurrent tasks whose activations are guaranteed to always be separated by no less than a minimum time span (also known as minimum inter-arrival time). Sporadic tasks are a key part of the automotive case study in PROXIMA.
Erika does not natively support sporadic tasks, but it offers a timing protection system to prevent a task from being released before a given time span. This mechanism, however, is not yet implemented for the Tricore owing to the lack of hardware support.
To circumvent this limitation, we devised an alternative implementation out of a combination of periodic activation and conditional wait. Listing 15 provides a basic example of a sporadic task realised in that manner. A cyclic alarm periodically activates `Task1` that immediately suspends itself on event `Event1`. Only when `Event1` is issued by `Task2`, `Task1` gets ready again. In this way, `Task1` cannot be activated more frequently than its minimum inter-arrival time.

Listing 15: code.c: Sporadic task

```
#include "ee.h"
volatile EE_UINT32 counter1 = 0;
volatile EE_UINT32 counter2 = 0;

TASK(Task1){
  WaitEvent(Event1);
  ClearEvent(Event1);
  counter1++;
  TerminateTask();
}

TASK(Task2){
  SetEvent(Task1, Event1);
  counter2++;
  TerminateTask();
}

int main(void){
  StartOS(OSDEFAULTAPPMODE);
  return0;
}
```

The corresponding task configuration is reported in Listing 16. Events used by tasks must be declared in the OIL within the same scope of the task: the bit-mask associated to the event is automatically generated in this example. Each task that is expected to wait for an Event must declare it as a task property and use a private stack (hence an ECCX conformance class is required).

Listing 16: Configuring a sporadic task

```
TASK Task1{
  PRIORITY = 2;
  AUTOSTART = FALSE;
  SCHEDULE = FULL;
  STACK = PRIVATE{
    SYS_SIZE = 2048;
  };
  EVENT = Event1;
};

TASK Task2{
  PRIORITY = 1;
  AUTOSTART = FALSE;
  SCHEDULE = FULL;
  STACK = PRIVATE{
    SYS_SIZE = 2048;
  };
};

EVENT Event1{
  MASK = AUTO;
};
```

2.2.4 Time-Composable Implementation

The property of time composability takes full meaning in single-core processor settings, to signify that the system is composable in the time domain; that is, the timing behaviour of a system component (an application) does not suffer the presence of other components. The key role of a time-composable RTOS in supporting the incremental development and verification of applications on modern processor architectures is well acknowledged [1].

Time-composability is generally hard to achieve, especially on multi-core processors, where parallel contention of shared hardware resources is the natural behaviour of the system, and in the absence of hardware modifications, which cannot be assumed for COTS processors requires changing the way RTOS services operate on the underlying hardware and are made available to user applications. In some cases, RTOS capabilities are severely limited, to meet time-composability requirements as well.

The original code-base in Erika assumes by default a strictly partitioned multi-core configuration, where each core is associated to a system partition. Partitioning eases the process of making Erika time-composable as it allows the system to

be regarded a set of partially independent single-core systems with inter-partition communications as the main source of interference determined by software characteristics of the application. For this reason, we first modified core Erika services to be time-composable within the core, similarly to what we did in [1], then we added a time-composable inter-partition communication support, which relies on an AUTOSAR IOC component we modified to use a TDM-based scheme to share the access to the LMU memory through the crossbar.
For this Erika-TC (for time-composable) release we injected time-composability into core Erika services, in terms of zero-disturbance[5] and steady timing behaviour[6]. In the following, we provide an overview of the Erika RTOS features that we rendered time-composable. Each feature is presented with particular attention to configuration guidelines and TC assessment results.

Deployment configuration In the Aurix processor, configuring the system to be deployed on specific memory segments is fundamental to guarantee zero-disturbance. As the Tricore hardware layout allows keeping a large amount of information in scratchpad memory, we force Erika code and data to be entirely deployed into them, thus excluding any cached memory areas. Erikas memory footprint is sufficiently small and this solution does not seem to be significantly restrictive. To enable this configuration, the EE EXECUTE FROM RAM option in the OIL is required, as shown in Listing 17.
The default Erika configuration tries to deploy also the user applications to the scratchpads. In some cases, however, applications might be deployed to different memory segments as, for example, they might not completely fit into the scratchpads. For example, options EE ENABLE TASA SUPPORT and EE TASKS ON PFLASH offload applications code to the PFlash cached segment.

Listing 17: Deploying to scratchpad memories

```
EE_OPT = "EE_EXECUTE_FROM_RAM";
```

Run-to-completion A simple but effective solution for minimising the disturbance effect of OS services and inter-task interference consists in enforcing run-to-completion execution semantics. Under this scheme, application jobs cannot be interrupted during execution and all scheduling decisions are deferred until job completion, even when higher priority tasks become ready for execution. Run-to-completion used non-preemptive scheduling, which is already supported by the Erika code-base, but which – for use in PROXIMA – also required removing any asynchronous source of interference originated, for example, by the tick-based time management whose disturbance breaks time composability.
In PROXIMA, to properly support time-composable run-to-completion semantics, we removed the tick-based scheduling in favour of a timer-based scheduling with deferred timer interrupts. In addition to that, we re-implemented some scheduling

[5]Erika services must not in uence the application in presence of hardware with a history-dependent behaviour and because of time-triggered actions (i.e., the tick management) whose timing behaviour is not constant.

[6]The jittery timing behaviour of Erika services, caused by the hardware/software status and input data, must be avoided.

primitives to make execute in constant time. We discuss both features later in this section.
The non-preemptive scheduling support implemented in Erika includes support for the priority ceiling protocol. Each task is therefore assigned a ready (static) priority and a dispatching (dynamic) priority. The latter is assigned to make tasks non-preemptive, so that they run at a priority higher than all premptable tasks. In practice, this is obtained by setting a system ceiling value to the dispatching priority, which causes the CPU to be released only at job completion. Non-preemptive tasks are characterised by the `SCHEDULE` parameter set to `NONE`, or by its absence, as shown in Listing 18. The `FULL` value, instead, designates fully-preemptive tasks.

Listing 18: Run-to-completion task specification

```
TASK Task1{
  CPU_ID = "CPU0";
  PRIORITY = 1;
  AUTOSTART = FALSE;
  SCHEDULE = NONE;
  STACK = SHARED;
};
```

We identified two situations where tasks should be non-preemptive: when they access cached memory areas (i.e., the cached PFlash), or when they use inter-core communication primitives. In all other cases, tasks can be preempted without breaking time-composability.
Non-preemptive scheduling might penalise the schedulability and responsiveness of the system. For this reason, non-preemptive execution should be as short as possible.

Constant-time scheduling We devoted significant effort to re-implement Erikas scheduling primitives to make them use bitwise operations on bit-masks to achieve constant-time execution behaviour in all of the four OSEK conformance classes.
All conformance classes now use a 32-bit mask as ready queue (`EE rq bitmask`). Scheduling decisions are made on them on fixed-priority basis, with the following pre-class provisos:

- for BCC1 and ECC1 classes, each task is assigned a priority value that corresponds to a bit in the ready queue mask, so that all tasks have distinct priorities (obviously limited to 32 on the Aurix). Two arrays are used to map the task to the corresponding priority value (`EE th ready prio`) and vice-versa (`EE prio link`). The same principle used to set tasks in the ready queue has been also used to keep track of suspended tasks;

- for BCC2 and ECC2 classes, a task queue is associated to each priority level, so that, multiple tasks can be at the same priority level and pending task activations are allowed. When a task becomes ready, it is stored in the corresponding queue and the bit corresponding to its priority level is set on the ready bit mask. The scheduling policy applied to these conformance classes is FIFO within priorities.

This representation allows the ready queue to be updated in constant-time using bitwise operations. The task selection, for example, requires identifying the most significant bit in the ready queue bit-mask. To do so, we exploited the perfect

hashing of De Bruijn sequences [15]: the `CLZ` instruction of the Aurix provides the same semantics, but cannot guarantee constant-time latency.
To enable constant-time scheduling primitives, the `EE SCHED O1` option have to be set in the OIL, as shown in Listing 19. When one of the BCC1 or ECC1 classes is selected, the `EE prio link` has to be declared, as shown in Listing 20. This array must contain tasks pinned to the specific core, at increasing priority order. The first priority level is reserved to the background task and the `EE NIL` value is required for it.

Listing 19: Constant-time scheduling option

```
EE_OPT = "__EE_SCHED_O1__";
```

Listing 20: code.c: Priority level to task mapping array

```
#ifdef __EE_SCHED_O1__
#define EE_MAX_PRIOEE_MAX_TASK + 1
const EE_TIDEE_prio_link[EE_MAX_PRIO] = {
  EE_NIL,
  Task1,
  Task2,
  // ... map priority level to task
};
#endif
```

So far, we presented how a single update operation, that is, the insertion of a task in the ready queue and electing the next task to be dispatched, has a constant execution time. As tasks are triggered by different alarms, however, the number of update operations on the ready queue depends on the number of alarms expiring at the same time. Moreover, alarms are maintained in an ordered list and each expiration times is expressed as the time span from the immediately preceding expiration time. Updating that list may take a variable amount of time, depending on the list state: inserting an alarm needs to find the correct position in the list. Cyclic alarms would allow determining a hyper-period and thus a recurring time-expiration pattern that would permit to avoid the run-time update of alarms. This solution, however, is rather restrictive. Our re-implementation does not use this scheme and provides more flexibility. Therefore, we only partially achieve the desired steady execution time behaviour for all our scheduling primitives. Our choice reflects the needs of the automotive use case in PROXIMA, where the number of cyclic tasks is significantly lower than the number of sporadic tasks, which causes alarms to have to be maintained in an ordered list.

2.2.5 Evaluation by PAKs

To gauge the gain obtained in terms of time-composability by our modifications, we evaluated the timing behaviour of the revised scheduling primitives, with PAKs, before releasing Erika-TC for use in the automotive use case. A PAK (short for PROXIMA Application Kernel) is an ad-hoc application program designed to stress specific parts of the system. PAKs have been specifically designed to compare the performance of scheduling operations, such as inserting a task into the ready queue and computing the context switch, of the original Erika release to those of the time-composable one.
In our evaluation experiments, six tasks are assigned to the first core and one task on the second core. Tasks have the same period, but their first activation is offset by one second from each other: the second task starts one second later than the first; the third starts one second later than the second; and so forth.

Figure 14 reports the cost of the task activation primitive, where the original Erika implementation was incurring much jitter.

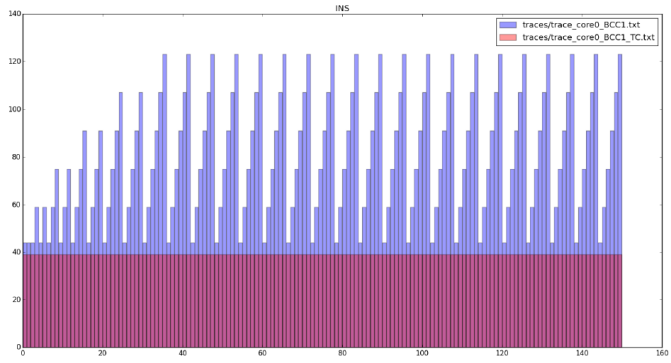

Figure 14: Task activation with insertion into the ready queue (processor set to 100 Mhz; kernel set to BCC1 configuration)

Figure 15 shows instead the cost to compute the context switch, between the task atop of the stack and the highest priority task on the ready queue.

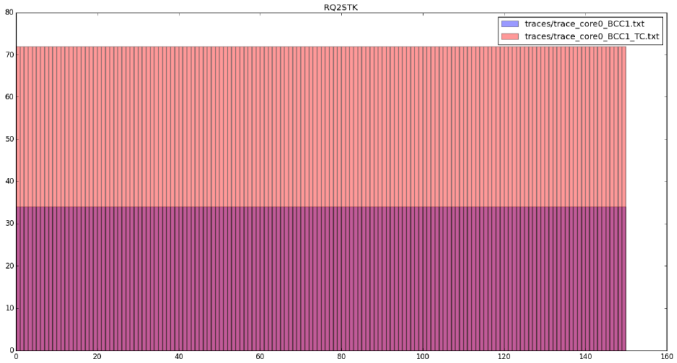

Figure 15: Election of the next task to dispatch (processor set to 100 Mhz; kernel set to BCC1 configuration)

Our time-composable primitives behave as good as expected. Significant variations instead are incurred on task activation when the original insert primitive is used. Those variations are caused by the implementation of the ready queue, which requires tasks to be ordered when inserted. Conversely, constant execution time can be observed using the time-composable implementation, which uses only bitwise operations. No variations, instead, have been detected in the context switch, as popping the task is an O(1) operation by nature.
Interestingly, in force of partitioned scheduling and no preferential use of scratchpad memories in place of shared cacheable memory, all activity in both experiments occurs within the designated core, no inter-core interference is incurred.

Events Events are a fundamental RTOS feature for automotive applications. Unfortunately, their implementation in the original Erika may break the principle of time-composability.

Consider for example, Listing 15, where `Task1` waits for `Event1` and `Task2` sets the corresponding event. `Task1`, which has higher priority than `Task2` executes until the `WaitEvent` primitive is called, which causes the task to move into a waiting state and calls for dispatching. `Task1` will be resumed as soon as `Task2` calls the corresponding `SetEvent` primitive. When this happens, `Task1` resumes its execution and immediately preempts `Task2`, owing to its higher priority, thus breaking the run-to-completion execution semantics for `Task2`.
We therefore modified Erika to prevent the running task from being preempted after setting an event.
To enable deferred events handling, you need to set the `EE DEFERRED EVENT HANDLING` option in the OIL.
Interestingly, the sporadic task model presented earlier includes two distinct invocation events: the periodic alarm on the one hand, and the resume from the waiting state on the other. Having multiple invocation events may complicate timing analysis and also expose dependence on the periodic alarm. We therefore explored an alternative model for sporadic tasks, which uses a single invocation event only. The idea is to make the call to `WaitEvent` the single invocation point, so that the sporadic tasks revolves in an infinite loop around it, as shown in Listing 21. To that end, we use a bit mask consisting of a `TimingProtectionEvent` and the original `Event1` to resume `Task1` when all the corresponding bits are up. At that moment in time, the required minimum inter-arrival time has elapsed and `Task2` has called the `SetEvent` primitive.

Listing 21: code.c: Single-invocation sporadic task

```
#defineTASK1_MINIMUM_INTERARRIVAL_TIME500
TASK(Task1){
  for(;;){
    WaitEvent(TimingProtectionEvent | Event1);
    ClearEvent(TimingProtectionEvent | Event1);
    SetRelAlarm(Alarm_Task1, TASK1_MINIMUM_INTERARRIVAL_TIME, 0);
    /*startofthetaskbody*/
    counter1++;
   /*endofthetaskbody*/
  }
}
```

The OIL is updated accordingly. As shown in Listing 22, the alarm used to activate `Task1` now sets the `TimingProtectionEvent` event when the prescribed minimum inter-arrival time is elapsed, and is armed by the task by means of the `SetRelAlarm` primitive. To initially activate the task, the AUTOSTART options is set to TRUE as well as the first alarm expiration.
A similar approach should be used when potentially blocking `WaitEvent` calls are in the functional specification of the task. Such a task should be split into multiple sub-tasks, each of which characterised by a single `WaitEvent` call, as a single activation point.

Listing 22: Events configuration

```
TASKTask1{
  PRIORITY = 2;
  AUTOSTART = TRUE;
  SCHEDULE = FULL;
  STACK = PRIVATE{
    SYS_SIZE = 2048;
  };
  EVENT = Event1;
};

EVENT Event1{MASK = AUTO;};
EVENT TimingProtectionEvent{MASK = AUTO;};

ALARM Alarm_Task1{
  COUNTER = system_timer;
```

```
  ACTION = SETEVENT{
    TASK = Task1;
    EVENT = TimingProtectionEvent;
  };
  AUTOSTART = TRUE{
    ALARMTIME = 1000;
    APPMODE=OSDEFAULTAPPMODE;
  };
};
```

The model presented so far assumes that the waiting task is resumed for execution only when all events have been set. However, this semantic is not warranted by the original `WaitEvent` primitive. We therefore specifically introduced it to support the scenario where tasks need to be activated by multiple events. In this case, the timing protection mechanism might result unnecessary for tasks whose activation depends on other cyclic tasks. To enable the all semantics, you need to use the `EE CUMULATE EVENTS` option.

Time management Erika's timing services originally relied on a tick-based implementation whose disturbance effect on user applications could completely break time-composability (particularly with finer tick granularity).
To abate this source of interference, we redesigned the timing services to use less intrusive interval timers.
Interestingly, interval timers alone are not sufficient to this end, as a running application might still be interrupted by timer interrupts. To restore run-to-completion execution semantics proper in this case, we need alarm expiration events to not interrupt the running application.
To this end, we defer all timing events, that otherwise would be triggered during execution, to right after the completion of the running job, that is until the next dispatching point. This modification ensures the task to run to completion, without being interrupted, and to keep the possibly jittery behaviour of scheduling primitives between two consecutive tasks execution. Notably, we apply this motification only to non-preemptive tasks: it is therefore important to mark each critical tasks as such in the OIL.

Inter-core communication Our proposal of time-composable inter-core communication builds on the AUTOSAR Inter OS-Application Communicator (IOC) layer, which we modified to support a software-enforced TDM-based arbitration mechanism for the Tricore crossbar. This time-composable inter-core communication framework is now part of the time-composable extension of Erika.
The details of the approach have been presented in [17], where we show how that mechanism can be exploited at the OS level to enforce timing isolation and to have predictable effects on the bus contention for all communications across applications which involves accessing a shared medium (e.g., LMU).
The original Erika implementation of the message-passing API has been extended to enforce a configurable arbitration policy by splitting the exchanged messages into chunks and arbitrate each chunk transfer in TDMA-like time slots. TDMA frame, slots and chunk size can be configured to better meet the application requirements.
To address specific needs of mixed-criticality systems, the cited work also proposes a more flexible TDMA-based arbitration, which allows reserving more than one communication slot to selected cores. This enhanced version of TDMA enables the provision of better bandwidth levels to cores running high-criticality applications.

Further details on the proposal and the experimental evaluation of it are reported in [17] (which is a public-access document) and not included here.

2.2.6 Using Erika-TC

We now provide instructions to set up a working environment and build your own applications on top of our Erika-TC code-base.
The PROXIMA release of Erika-TC includes the following artefacts:

- erika-tc@3274.patch: the patch to make Erika time-composable and to enable the support for TASA (see Section 3.3 in this document);
- Makefile: a makefile to build Erika and user applications;
- erika-TASA-c0 . . . erika-TASA-c3: four tutorial example configurations intended to support and evaluate TASA;
- erika-TASA-tasks, erika-TASA-events*: PAKs to assess the time-composable execution behaviour of Erika-TC;
- tc-test.py: a simple tool to analyze traces collected on Erika TC primitives;
- setEnv.sh, buildExps.sh: a script to set up the working environment and build the aforementioned examples projects.

In case you are running a Windows machine, please check you have both cygwin (`http://cygwin.com/`) and the Hightech Tricore toolchain (`http://free-entry-toolchain.hightec-rt.com/`) correctly installed on your host. All commands and scripts must be executed in the cygwin shell. Make sure at least `make`, `svn`, `unzip` and `wget` are available on your cygwin setup and your `PATH` environment variable points to the cygwin version of `make`.
A simple working environment can be automatically defined by using the `setEnv.sh` script provided. The following three basic steps are needed to set up your working environment:

1. obtain the official RT-Druid tool (`http://www.evidence.eu.com/erika-builds/ee_250/CLI/RT-Druid_2.5.0.zip`), which translates OIL specifications into Erika source code;
2. obtain the official Erika Enterprise release at revision 3274 (`svn://svn.tuxfamily.org/svnroot/erika/erikae/repos/ee/trunk/ee@3274/`);
3. apply the Time-composabiity patch `erika-tc@3274.patch` to the official Erikas source code.

In the following, we assume `WORKING_DIR` is the absolute path to your working directory and `APP_DIR` is the relative path to your application starting from `WORKING_DIR`. We also assume `erika-tc@3274.patch` and `Makefile` are in your working directory. The provided script `setEnv.sh` executes steps 1 to 3, taking `WORKING_DIR` in input.
At this point, a directory called `tools` has been created on your working path, which contains the Erika source code (`ee`) and the RT-Druid tool. You are now ready to compile your applications.

As a prerequisite to compilation, a working environment should be in place. Your application, typically comprising of an OIL specification file, several C source files and linker scripts, should already be in `APP_DIR` in your working path. The building process, briefly described earlier in this report, follows two steps. First, the system specifications described by the OIL need to be translated into source code to get an Erika configuration fit for your application and targeted to the Tricore architecture. Subsequently, the whole system can be compiled and linked together. Listing 23 shows the sequence of commands to configure and compile the application against Erika-TC.

Listing 23: Configuring and compiling your application

```
cd $WORKING_DIR
INPUTS_PATH = $APP_DIRBUILD_DIR=$APP_DIR/outputmakebuild
```

All the output produced will end to an output directory in your `APP_DIR` path, with a per-core organization. Depending on the system configuration, you will either get a single executable or an ELF file for each configured core.

3 Software Randomisation

Software Randomisation is the primary MBPTA enabler on COTS systems. Software randomisation mimics the behaviour of a random placement cache using software means. A random placement cache maps a specific piece of data in random cache lines across different executions, using a hardware random placement hash function. In COTS systems on the other hand, the hardware is not possible to be modified. Therefore the random placement of the data is achieved in an indirect manner, based on the observation that the deterministic placement function of the cache (usually modulo due to its simplicity) defines a fixed mapping from main memory addresses to cache lines. By randomly changing the position of memory objects (functions, stack frames, global data) in the main memory across different program executions, the said objects are mapped in random cache lines.
The difference between the hardware-based randomisation and software randomisation is the randomisation granularity: in the former it is on cache line basis, while software randomisation is performed on a coarser-grain, defined by the size of each individual memory object, which can span multiple cache lines. However, from MBPTA perspective this granularity difference is irrelevant, since in both cases the cache layouts have a probability of appearance and the MBPTA requirement inherited by EVT, execution times with i.i.d. properties is satisfied, as shown by passing the appropriate statistical tests.
In PROXIMA we have used two variants of software randomisation: dynamic and static. Dynamic Software randomisation (DSR) [10] [16] has been inherited from the PROARTIS project and is the primary software randomisation technique used in the project, since it has been been ported in two of the three COTS platforms we consider, FPGA COTS LEON3 and P4080. The dynamic software randomisation is named after its characteristic to modify the memory layout of the program dynamically at program execution.
However, the dynamic software randomisation is not applicable in all platforms, because it is based on self-modifying code, which is not supported by some architectures like the AURIX platform considered in the project [9]. For this reason, we have developed from scratch a static software randomisation variant named TASA [11], to address specifically those architectures. The difference of static software randomisation is that the memory layout is modified statically at compilation/linkage time as opposed to runtime.
Regardless of the variant, both software randomisation solutions are equivalent from MBPTA point of view, since both provide similar and trustworthy pWCET estimations. Regarding overheads and certification, which are further discussed in the Subsection 3.3.5, the static variant exhibits certain advantages over the dynamic variant. However, the dynamic software randomisation is currently in a higher readiness level compared to the static solution, due to the fact that it has been in use for 6 years and has been tested with multiple industrial case studies and ported in several platforms in both PROARTIS and PROXIMA projects. On the other hand, the static software randomisation despite its promising features, it has been developed the last couple of years and only tested on a single case study which is enough for its validation as a proof of concept, but not ready yet for industrial use.

In the following sections we examine the latest advances in the development of the software randomisation solutions employed in each of the target platforms, based on the feedback received from the WP4 users. Also we provide the evaluation of the latest versions of software randomisation on the corresponding platforms.

3.1 FPGA COTS

The dynamic software randomisation inherited by PROARTIS has been first ported in the FPGA COTS LEON3 platform. In D2.7, Chapter 3 from the last reporting period we have provided a detailed explanation about the internals of dynamic software randomisation as well as the decisions taken to perform the porting on the SPARCv8 based LEON3 platform.
For convenience we provide a short description below, in order to the reader to be able to relate with the updates performed in the refined and final implementation of dynamic software randomisation.

3.1.1 High Level Description of Dynamic Software Randomisation Port at m18

The dynamic software randomisation comprise a combination of a compiler pass and a runtime system. The compiler pass creates metadata for the memory objects which will be relocated at runtime and modifies appropriately the code of the software in order to allow stack randomisation. At program execution, the runtime system uses the metadata generated during the compilation in order to know their original location and sizes and moves them to new random locations in the main memory. Also it updates their old locations to ensure that the memory objects are accessed using their new positions, which in the case of code randomisation requires self-modifying code. Consequently, since the target system implements a Harvard architecture (separate instruction and data caches) a cache flush needs to follow the relocation of each function.
The relocation scheme employed by the original implementation of software randomisation and its initial port to LEON3 at m18 is *lazy relocation*. This means that a function is relocated only if it is called at runtime. In order to detect and initiate this event, in the LEON3 port the first instructions of each function are replaced at initialisation time with a code sequence which invokes the relocation routine, while a copy of the original replaced instructions is retained. Therefore during the first invocation of the function, the relocation function is called which subsequently allocates dynamically memory in a random position the code of the function is copied together with the copy of the original contents of the replaced instructions and the caches are flushed in order to guarantee that changes are visible. Finally the old location is replaced with a code sequence which redirects the control flow to the new function location, and the function is finally called from its new position.
The dynamic software randomisation uses a random memory allocator based on Heap Layers and DieHard [2] to ensure that each memory allocation is mapped in a different place. Also, the size of each function allocation is extended by an instruction cache way size (4KB) so that a random starting memory position can be selected inside the allocated memory chunk rounded to cache line boundaries up the first 4KB. This way we ensure that the starting position of the relocated

function can be randomly mapped in any cache line of the 4KB instruction cache. In order to select a position randomly in the allocated chunk of memory, we use a software implementation of the Multiply-With-Carry (MWC) random number generator, which is the same with the one implemented in the hardware randomised LEON3.
By m18 the DSR was already implemented to work on bare metal configuration as well as integrated with the research oriented real-time operating system RTEMS SMP variant explained in Section 2.1, which was the operating system of choice in the preliminary evaluation of software randomisation presented in the previous reporting period in D2.7.
In the following months, the implementation effort had been shifted over improving the identified limitations of the early implementation of DSR which is described in the next subsection, and supporting the industrial RTOSes which have been consindered for the case studies. In particular, DSR has been integrated with all PikeOS guest operating systems except Elinos whose use has been dropped early in the project: PikeOS Native, PikeOS APEX and PikeOS RTEMS.

3.1.2 Limitations of Dynamic Software Randomisation Port at m18 and their solutions

After the initial Dynamic Software Randomisation port has been released and tested by the WP4 industrial users, the following limitations have been identified and solved.

3.1.2.1 Relocation Scheme:

The initial port implemented a lazy relocation scheme, similar to previous implementations of Dynamic Software Randomisation. The original implementation of Dynamic Software Randomisation inherited by PROARTIS, was based on the Stabilizer [6] tool, designed for use in desktop and high-performance computers, which have significantly different requirements than real-time systems. In those systems, applications are linked with several libraries containing large amount of functions, while the application is actually using few of them. Therefore randomising a function only when it is called, allows to reduce application start-up times and additional memory overheads for large amounts of otherwise unused functions. However, the fact that the first invocation of each randomised function invokes its relocation complicates the worst case analysis of the software.
The worst case analysis can be divided in two parts: 1) timing and 2) memory consumption.

1. Industrial hard real-time applications have a periodic timing behaviour, which involves calling tasks (functions) repeatedly in specific time intervals. In avionics and automotive applications exist multiple periodic layers consisting of Major and Minor Frames, which allow to implement complex scheduling schemes. Thus, in such organisations one task may be invoked in every minor frame, while another task may be invoked only on a specific minor frame or frames.

   The worst case timing analysis of each task is required in order to ensure that all the tasks scheduled inside a minor frame fit in its period or whether

the frame period needs to increased or a new schedule needs to be derived. However, the first function invocation of each randomised function includes also its relocation which is an expensive operation followed by a cache flush as explained in the previous section. Therefore, the worst case execution time of each function is the one that includes the relocation and is very pessimistic, while all the following function invocations exhibit significantly lower execution times. Moreover, due to the complex hierarchical scheduling implemented in industrial systems, avoiding to consider the first invocation of each function is not trivial, since a minor frame may contain none or multiple tasks executed for the first time. Finally, it can be the case that a single minor frame contains several tasks executed for the first time, so that the first period of this frame results in an overrun causing the application to misbehave.

However, the implementation of an *eager* relocation policy solves the above problems. In eager relocation, all functions are relocated *before* the program execution, therefore the measured execution times are only affected by their memory layout as it is the case in hardware randomisation.

In order to implement this feature, we introduced an initialisation function that the user calls before starting executing the partition of his application. This function, relocates all functions and patches their original locations with an instruction sequence that redirects the control flow to their new location. Therefore once initialisation is over and the function is called, it is executed from the new location.

2. Besides timing, hard-real time systems require also guarantees about memory consumption. In fact, the main reason why dynamic memory allocation is disallowed in critical real-time systems, is that there is no guarantee that at the time of request there will be free memory in the system, compromising the correct execution of the software.

   The Dynamic Software Randomisation is based on dynamic memory allocation in order to obtain memory at random main memory positions to effectively randomise the application's memory layout. Earlier in this section we described the complex structure and behaviour of modern industrial real-time systems. Although the number of randomised functions and their size is fixed for a given software, in order to verify that the provisioned memory pool for dynamic memory allocation is sufficient at least a full major frame needs to be executed, to trigger each function at least once, so that it is relocated. Moreover, some leaf functions invocations might be data driven, and therefore hard to be exercised unless the application is executed for long time. As a consequence, lazy relocation complicates dynamic memory bounding.

   However, this problem no longer exists with our eager implementation in the final version for dynamic software randomisation. In that case, the memory pool provisioning can be validated very quickly, since all relocations take place at initialisation before the application is actually executed.

3.1.2.2 Dynamic Memory Consumption

As already mentioned, Dynamic Software Randomisation is based on dynamic memory allocation to obtain memory space for the relocations. The early LEON3 port as well as the original implementation from PROARTIS, was based on Stabilizer tool, which uses the Heap Layers and DieHard memory allocator to ensure random memory allocations. These memory allocators are specifically designed for the desktop and high-performance computing, with focus on high average performance. In fact, multithreaded applications are abundant in this domain, therefore these memory allocators are designed to provide fast and thread-safe allocations, while keeping low fragmentation. This, however, comes at the expense of increased memory consumption, which is not an issue in those systems since they are usually equipped with large physical memories and their operating systems provide abstractions such as swapping and lazy paging, which mitigate the problem.
On the other hand, real-time systems have different characteristics and needs. Real-time systems, like the ones used in the avionics domain, are memory constrained and their operating systems do not provide swapping. In particular, the features provided by hardware memory management units are only used in order to provide memory segregation, while each physical memory page is used by a single virtual page. Therefore the memory consumption is very important to be kept low, otherwise certain applications may be impossible to be executed. This was the case with the bigger (FCDC) of the two AIF applications, where while the application had code size of 5MB a memory pool of 256MB was unable to satisfy this requirement. The problem comes from the fact that the Heap Layers maintain separate memory blocks lists for each of the power of two sizes. When memory is requested and the list of the corresponding size is empty, the allocator reserves more memory from the pool, in order to populate the corresponding list. Each list is populated with 8MB of memory, which is expanded with an equal size when it is exhausted.
If the application contains a big number of functions and the functions have many significantly different sizes, this can cause several lists to be initialised exhausting quickly the provisioned memory pool.
In order to mitigate this problem, we have decided to replace the memory allocator used in the dynamic software randomisation with a simple bump allocator, which allocates memory chunks in a single contiguous memory pool. This way the memory requirements of DSR are reduced significantly.
Despite that the bump allocator is known to suffer from fragmentation problems, in the real-time domain and in our specific application this problem does not exist. This is because the allocated memory is never freed after allocation for the entire lifetime of the application, since the relocated functions will always continue to be executed from their new positions.
We have to note that the extra 8KB per function allocation is still retained in the new implementation, since it is the only way to guarantee that the relocated function can be allocated randomly in any cache line. Therefore the total memory consumption for code is the number of functions multiplied by 8KB.

3.1.2.3 Random Cache line Selection:

DSR is using the Multiply-with-Carry (MWC) random number generator to randomly select the cache line where each relocated function will be starting. This

random number generator has a very long period, 2^{60}, which is extremely long. To appreciate how big this number is, assuming a processor operating at 1GHz and 1 random number generated per cycle, the random number sequence would take 36 years to repeat.
However for the FPGA setup, out of the 32 generated bits, we only need to select 7 in order to select to which of the 128 cache lines of the instruction cache the function will be mapped to. In the initial implementation, the lowest 7 bits of the random number generator were selected. This resulted in a short period and a repeated pattern of the selected cache lines.
After an analysis we have changed this design in the final version to use the 7 upper bits of the random number. This way, the properties of the prng are retained as validated with appropriate tests for PRNG quality assessment as presented in detail in subsection 3.1.3.1.

3.1.3 Evaluation

In this section we present the evaluation of the final version of the Dynamic Software Randomisation. The improvements over the early DSR release that were described in the previous sections have been backported to all the PROXIMA considered RTOSes whih are supported on the FPGA platform, both industrial and research oriented. However, for the evaluation presented in this section, we only use the PikeOS Native configuration, to balance the fact that in the previous reporting period the early evaluation has been performed with the research oriented RTEMS SMP. Nevertheless, our benchmarks do not make use of any RTOS services, therefore the results regarding the application analysis are valid across any RTOS supported by DSR on the FPGA platform.
Note that in this deliverable we present only results with benchmarks, in order to provide evidence that the software randomisation works and evaluate its different aspects. Results with the case studies are presented in D4.8, while time composable execution time estimations for multicore configurations using software randomisation are presented in D3.8.

3.1.3.1 Cache line Randomisation

Pseudo-Random Number Generator (PRNG) design

The PRNG we use in our system to generate a random value in order to select randomly the cache line in which each memory object will be placed, is based in the Multiply-With-Carry (MWC) designed by George Marsaglia, using 1103515245 as multiplier and 12345 as carry. Since the number of possible cache sets in the cache is small, we perform a shifting, obtaining the 8 higher bits form the resultant value. The implementation is as follows:

```
unsigned int random_value()
{
        random_seed = (random_seed*1103515245+12345)&0x7fffffffU;
        return random_seed>>24;
}
```

Finally, we use the lowest 7 bits, in order to select randomly one of the 128 (2^7) possible cache sets.

The NIST Statistical Test Suite

To be able to analyse the randomness of our PRNG, we use the Statistical Test Suite (STS) provided by the National Institute of Standards and Technology (NIST). The NIST Statistical Test Suite (NIST-STS) is a statistical package consisting of 15 tests that were developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudorandom number generators. Specific tests for the case of a PRNG in the context of safety-relevant software do not exist. However, cryptograpy is a field with very stringent randomness requirements, so we inherit randomness tests from that field at the risk of imposing higher constraints than needed on the PRNG. These tests focus on a variety of different types of non-randomness that could exist in a sequence. Some tests are decomposable into a variety of subtests. The 15 tests are:

1. The Frequency (Monobit) Test
2. Frequency Test within a Block
3. The Runs Test
4. Tests for the Longest-Run-of-Ones in a Block
5. The Binary Matrix Rank Test
6. The Discrete Fourier Transform (Spectral) Test
7. The Non-overlapping Template Matching Test
8. The Overlapping Template Matching Test
9. Maurer's "Universal Statistical" Test
10. The Linear Complexity Test
11. The Serial Test
12. The Approximate Entropy Test
13. The Cumulative Sums (Cusums) Test
14. The Random Excursions Test
15. The Random Excursions Variant Test

An extended explanation of the above tests can be found in [14].

Methodology

The NIST-STS performs tests in bit streams, so we wanted to ensure that each bit stream has enough values and that we explored the required number of bit streams. With this in mind, we decided to provide 400000000 values, splitting them in 100 different blocks. Note that according to [14] at least 100 bit streams of 400000 bits each are needed for a correct application of all tests, and we provide 100 bitstreams

of 4000000 bits each. Also, the NIST-STS requires binary data, so the values used are saved in in a file as binary.
Once the NIST-STS is executed with the PRNG binary values as input, we get a file with the summary of each test. Each test shows the frequency of the P-Values, the P-value that arises via the application of a chi-square test, the proportion of binary sequences that passed and the corresponding statistical test. Each test has a minimum pass rate (96/100 for all the tests with the exception of the random excursion (variant) test which has 84/100), and the proportion of each test must be higher than the minimum.
Given this information, we analysed our PRNG values with the tool and we passed all the tests as shown in the following tables:

Name	Proportion
Frequency	98/100
BlockFrequency	100/100
CumulativeSums	98/100
CumulativeSums	99/100
Runs	99/100
LongestRun	99/100
Rank	97/100
FFT	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100

Name	Proportion
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100

Name	Proportion
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100

Name	Proportion
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
OverlappingTemplate	99/100
Universal	100/100
ApproximateEntropy	99/100
RandomExcursions	54/54
RandomExcursions	54/54
RandomExcursions	54/54
RandomExcursions	53/54
RandomExcursions	53/54
RandomExcursions	53/54
RandomExcursions	52/54
RandomExcursions	53/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	53/54
RandomExcursionsVariant	53/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	52/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54

Name	Proportion
RandomExcursionsVariant	53/54
RandomExcursionsVariant	53/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
RandomExcursionsVariant	54/54
Serial	98/100
Serial	97/100
LinearComplexity	100/100

Overall, we see that the quality of the PRNG used to generate the 7-bit inputs for software randomisation, and therefore random cache placements obtained, meets the highest standards.

3.1.3.2 Memory overheads

In this section we evaluate the memory overheads of the PROXIMA FPGA COTS board port of software randomisation. These memory overheads can be divided in two groups: static memory overheads and dynamic. The static memory overhead corresponds to the code and data increase due to DSR, and it is reflected in the binary sections which are loaded in the main memory before a program is executed. We remind the reader that an elf binary contains the following sections, as explained in the D2.7: *.text* which contains the program code and *.data* for the program data.
The dynamic memory overhead corresponds to the increased use of the memory occupied during the program execution, in the dynamically sized segments of *stack* and *heap*. Stack contains the local variables, arguments and return address of each function that is active in the function call tree, while the the heap contains data allocated dynamically. In the following we evaluate DSR overheads in these two aspects.

Static Memory Overheads

For the evaluation of the static memory overheads we have selected the AutoBench applications from the EEMBC benchmark suite, which are the common choice for evaluation in real-time embedded systems, since they resemble commonly used workloads in the automotive sector. For each of the benchmarks we present relative size increase with respect to the size of the corresponding memory section when DSR is not used.

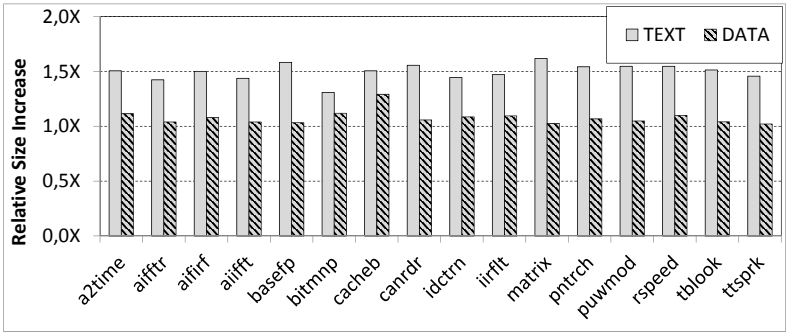

Figure 16: Memory overhead for different binary sections. Results are normalised to the binary generated without software randomisation.

In Figure 16 we can see the static memory overheads for the code and data binary segments. We notice that the code size is increased by 50% due to the introduction of the additional code to perform stack randomisation, as well by the runtime system

Table 5: Worst-Case Heap Usage in various CRAND configurations due to dynamic code allocations

# of Functions	Total Code Size (Bytes)	Total Allocated Size	Relative increase
4	14096	34576	2,45×
8	19824	56688	2,86×
16	25024	94656	3,78×

which contains the necessary code for code randomisation. Despite this increase seems considerable, in reality it is not.
The reason is that the majority of this increase comes from the fact that the runtime is linked with the randomised application. Since the original code size of the applications is in the range of 70-100 KBs, linking with a library of 45 KB causes the relative increase to look big. However, linking the library to an industrial application with size bigger that a MB, like the ones considered by the WP4 users in the corresponding deliverables, this overhead is negligible.
A small part of the code increment comes from the additional code introduced for performing the stack randomisation. This heavily depends on the stack usage of the randomised application, however the EEMBC benchmarks contain few functions without many local variables and for this reason this is very small compared to the total size of the executable. On the other hand, in an industrial application with many functions and heavy stack usage like the railway one, the code increase due to the code generated can be up to 80% as described in the deliverable D4.8.
Regarding the data segment, the increase is due to the introduction of metadata for each randomised function and the data required by runtime library. The size of metadata is proportional to the number of functions in the code, and since EEMBC benchmarks consist of few functions, the total increase is very small, than 7%. However an industrial application with thousands of functions might experience a larger increase in this segment.

Dynamic Memory Overheads

The dynamic memory overhead of DSR comes from the increased memory used to achieve random placement of memory objects. In particular, the increase in the maximum stack size is related to the fact that the stack frame of each randomised function is increased by a random amount up to the size each cache way. On the other hand, the increase in the heap comes from the dynamic allocation required for each function that is being randomised/relocated.
The main characteristic of dynamic memory segments is that in a real-time system need to be bounded. Special static timing analysis tools are used in order to compute the maximum stack usage, while the heap usage is prohibited in critical real-time systems. Since we don't have access to a static stack analysis tool we need to derive this information in an ad-hoc way. Due to the long a tedious process to infer this information from the binary and combine it with the maximum possible stack depth, we don't provide this information for the EEMBC benchmarks. Moreover, such analysis is beyond of the scope of this evaluation.
Instead we use the CRAND and SRAND benchmarks, introduced in D2.7 for the software randomisation evaluation, which stress code randomisation and stack randomisation respectively.
Recall that the SRAND is implemented as a recursive function, thus using the stack considerably. For the stack usage we obtain the maximum recursion depth by executing

Table 6: Worst-Case Stack Usage in various SRAND configurations due to stack randomisation

Max Depth	Original Stack (Bytes)	Randomised Stack (Bytes)	Relative increase
6	768	13371	17,41×
26	3328	57941	17,41×
51	6656	115882	17,41×

the executable a single time. Then the original stack frame is extracted from the binary and it is increased by worst case stack usage (4KB), to account for the worst case scenario that in each function invocation the maximum padding is (randomly) selected. Then the two factors are multiplied together to compute the worst case memory increase.
Table 6 shows the relative increase between the overall worst case stack consumption for various configurations of SRAND, compared to the default stack consumption when randomisation is not used. We can see that the total relative overhead of stack randomisation is constant (17×) regardless the maximum stack depth. The reason for this is that in both randomised and non-randomised binaries the total stack usage is proportional to the maximum recursion depth, therefore the relative increase is only related to the individual stack frame increase of the function in the binary. Note that this individual increase is slightly bigger than exactly 4KB, because the sw randomisation compiler pass causes a slightly different code to be generated, with a different number of register spills due to the extra operations needed to manipulate the stack at runtime. However, the total increase is dominated by the necessary 4KB increase at most, which is needed to guarantee that start of the stack of each function can be mapped to any data cache line. It is important to recall that this overhead is not actually the size of the stack which will be consumed at runtime, but its worst case bound. In fact, since each function invocation is going to have a random padding in its stack between 0 and 4KB, the actual stack size required only a fraction of it. However, due to the conservative nature of hard-real time systems, we have to consider the worst case in our analysis.
For the heap we use CRAND since it contains several functions in order to stress the code randomisation. In this case, the memory allocator is instrumented in order to provide information about each allocation. Due to the facts that in this refined version of DSR we a) implement a lazy relocation scheme, b) we replaced the advanced memory allocator with a bump-allocator and c) we don't use dynamic memory allocation in the application (similar to a critical real-time application) we can obtain the maximum heap relatively easy with a single program execution.
In Table 8 we explore the effect of DSR in the dynamic memory consumption due to code randomisation, using various configurations of the CRAND micro-benchmark. We can see that the dynamic memory footprint of the program is increased linearly with the number of functions. The reason for this is that each function allocation is expanded by 4KB in order to be able to place the function in any instruction cache line.
Note that the dynamic memory consumption for both stack and heap account for the worst case scenarios. For this reason, the overheads of DSR are significantly high. However, in both code and stack randomisation cases, the 4KB size is the minimum size that can be added in order to achieve complete cache randomisation for instruction and data caches, equivalent to the hardware randomisation scenario. Therefore, there is a clear trade-off between the memory consumption and the ability to apply MBPTA without hardware modifications, when DSR is compared against hardware randomisation. Also when a system analysed with MBPTA enabled by DSR is compared with a system with-

out randomisation analysed by a conventional timing analysis method such as MBTA, the memory consumption is the price to pay for being able to use a simple method as MBPTA and obtain pWCETs that are insensitive to significant sources of jitter like the memory layout. In fact, not only the memory layout impact is factored out from the uncertainty present in the case of conventional analysis, but also in some cases DSR is able to improve both average and worst case performance, by reducing the probability of the software to experience poorly performing memory layouts, as we show in Subsections 3.1.3.4 and 3.1.3.5 later on.

3.1.3.3 Eager vs Lazy relocation

In this section we evaluate the difference between the Eager and Lazy relocation schemes regarding the WCET. For this purpose we use the CRAND benchmark.

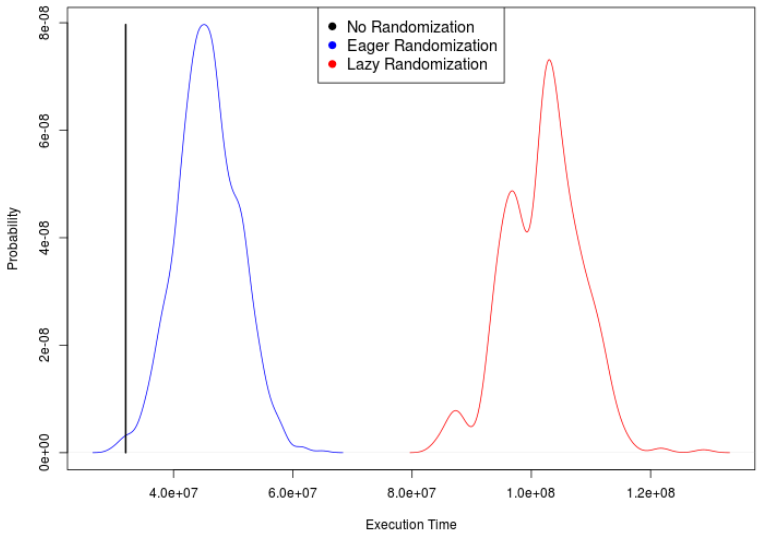

Figure 17: Execution time distribution for lazy and eager relocation software randomisation schemes, and for no software randomised one. The benchmark used is CRAND with 16 functions, with total size 125% of the cache size.

The Figure 17 shows the comparison between the execution time distribution of the two cases, using the CRAND benchmark with 16 functions and total size 125% of the instruction cache size. The first observation is that the relocation scheme does not affect at all the quality of the execution time randomisation, since in both cases the execution time becomes a random number distribution, close to a normal, with a significant variation. The variation exhibited in the execution times is clearly the effect of the randomisation of the memory layout, which effectively randomises the cache conflicts. On the other hand, the exact position of the distribution depends on the overheads of each relocation scheme. For this reason, as expected, the Eager relocation provides a lower execution time distribution, because in the observed execution times of each function, we don't account for the longer first invocation which includes the expensive relocation and cache flush operations. In particular, this difference causes the total execution time to be almost halved.
Comparing the average performance of the randomised versions with the non-randomised one we can see that for this configuration the possible execution times of the lazy re-

location more than 2× larger than the non-randomised version. On the other hand, the eager relocation provides execution times in the range of the of the non-randomised version with an expected slowdown of 20% due to the randomisation overhead of each function rediction to its new location.

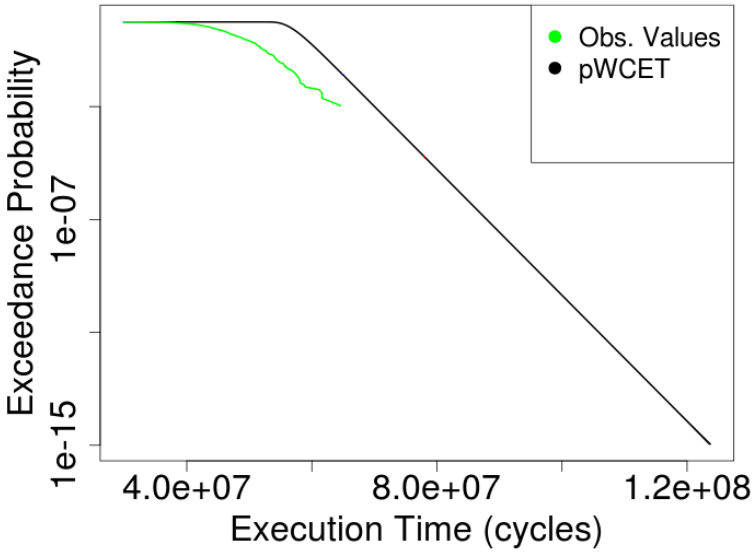

Figure 18: pWCET estimation for Eager Relocation.

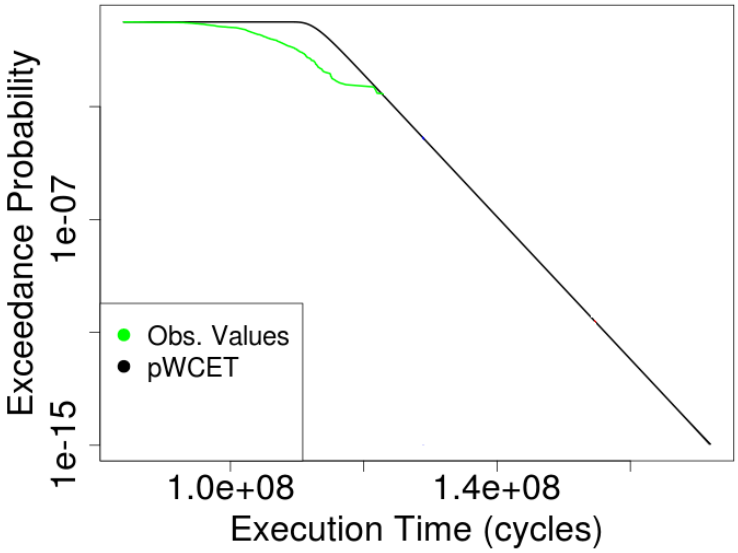

Figure 19: pWCET estimation for Lazy Relocation.

Note however that the execution time of the non-randomised version is provided for the exact memory layout for the binary used in testing. If this memory layout is changed (eg due to the incremental development process followed by the industry or final integration with other libraries and operating system) this execution time can change drastically, as shown by the variation of the random distributions. Therefore, while the randomised versions have an everage performance slowdown, they offer the advantage of taking into

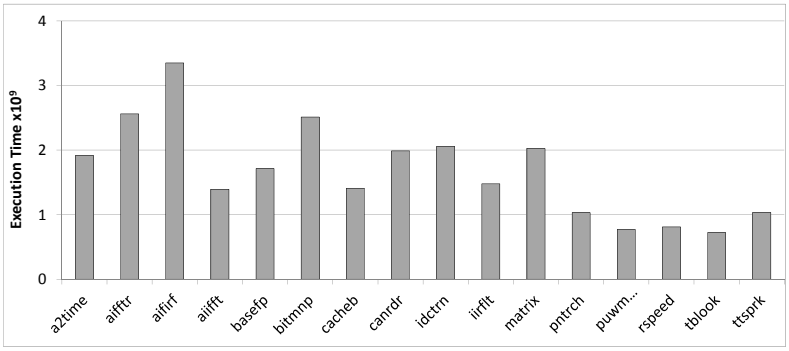

Figure 20: Worst Case Execution Time Estimations for DSR

account the effect of the memory layout in the execution time, which is further used to compute the pWCET. This way, the pWCET obtained with software randomisation are robust in memory layout changes, while WCET of a non-randomised one is heavily dependent on the actual memory layout.
In Figures 18 and 19 we present the pWCET estimations for the eager and lazy schemes of the software randomisation versions of the same application. Also we can compare those pWCETs with the current industrial practice of using the Maximum Observed Execution Time (MOET) increased by an engineering margin of 20% as a WCET bound. In eager relocation we observe that the obtained value for the pWCET up to the exceedance probability of 10^{-15} is significantly lower than the industrial practice without randomisation, which has a MOET of 1.3×10^8 and is increased to 1.56×10^8 with a 20% margin, resulting in a 21% improvement over the current industrial practice. On the other hand, the pWCET of the lazy relation is 8% larger than the industrial practice for exceedance probability 10^{-15} and only 2.5% larger for 10^{-12}. Therefore, even the lazy relocation scheme is competitive with respect to the solutions currently used in industry, while the optimised eager relocation is far superior.

3.1.3.4 pWCET Estimations

In the deliverable D2.7 during the previous reporting period, we presented results with our micro-benchmarks CRAND and SRAND. Instead of repeating the same experiments with the updated version of DSR, we use the EEMBC benchmark suite.
For each benchmark we collect 1000 execution times and we make sure that the collected execution times pass both the independent and identical distribution tests. Subsequently we generate the pWCET for each benchmark.
Figure 20 shows the results of the pWCET estimates for each benchmark using a 10^{-15} cut-off probability.

3.1.3.5 Average Performance

In this section we evaluate the effect of DSR compared to the average execution time, when DSR is not applied. As we have explained in the previous section, the execution time overhead of software randomisation comes from the fact that the an indirection is placed from the previous function location to the new one and that the stack randomisation code performs extra manipulations in the stack frame.
In Figure 21 we can see a comparison between the average performance of the EEMBC benchmarks with DSR as well as when they are executed without randomisation.
In general we see that that in the majority of the benchmarks the DSR causes a small degradation in the execution time less than 15%, with the exception of `aifftr` which

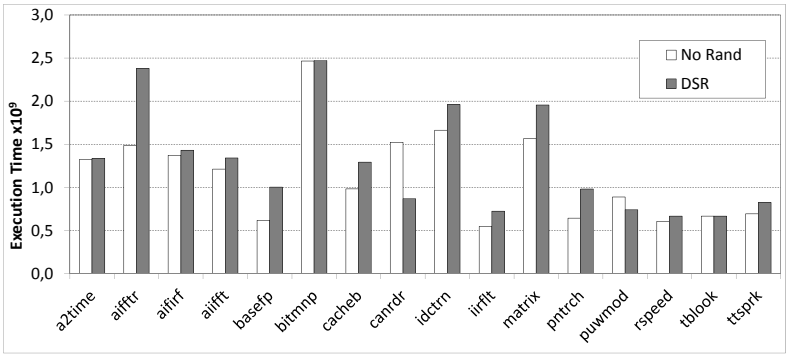

Figure 21: Average Execution Time comparison for DSR

has a 60%. We notice that the benchmarks which show a slowdown are the ones based significantly in floating point arithmetic such as `aifftr`, `aifirf`, `aifft`, `basefp`, `idctrn`, `matrix`. This is because the compiler pass which performs the randomisation increases the number floating point accesses, an event that has also been identified in the railway case study.
On the other hand, the integer-based benchmarks (`a2time`, `rsspeed`, `tblook`) have smaller slowdowns for the same reason. However, there are two cases (`canrdr` and `puwmod`) that DSR performs better than without randomisation. The reason for this is that the default memory layout created by the compiler happens to belong to the small set of "bad" layouts, which have a high number of conflicts in the cache. When software randomisation is used, each program execution results in a different memory layout. Since the majority of the cache layouts have less conflicts that the unfortunate one selected by the compiler, the average execution time is reduced by 60% in the case of `canrdr`, even with the additional overhead introduced by the redirections.

3.1.3.6 Event distributions in isolation

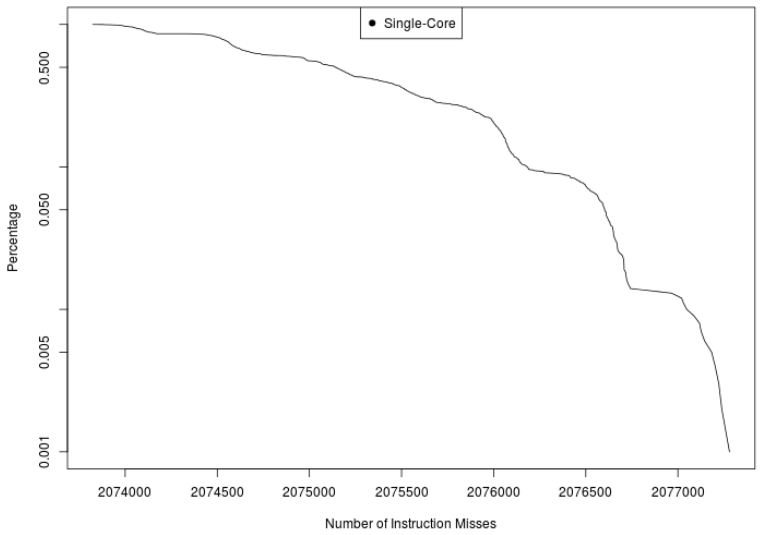

Figure 22: Instruction miss distribution for the configuration 6 (16 functions, with total size 125% of the instruction cache size.)

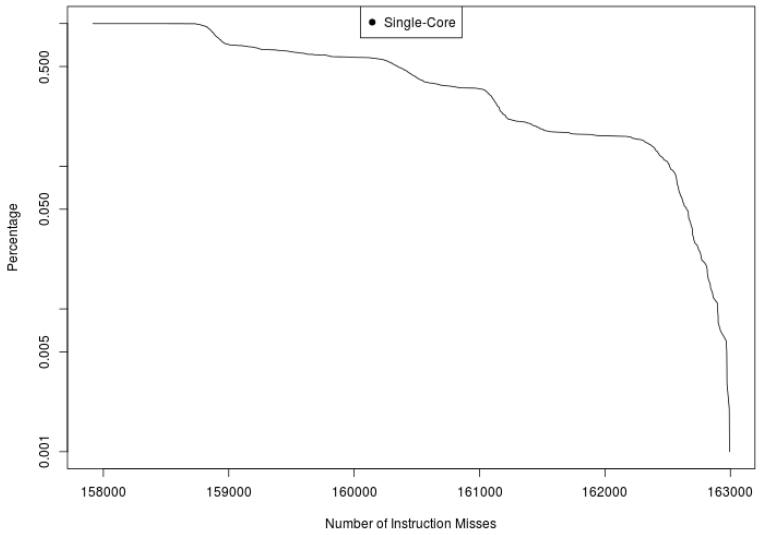

Figure 23: Data miss distribution for the same configuration.

In this subsection we provide evidence that software randomisation works as expected, by providing information about each event of interest. In the hardware configuration of the FPGA which is used for the evaluation, we have four cores with private L1 caches and a partitioned shared L2, interconnected with a bus with round-robin arbitration policy. In the single core configuration the events of interest are the L1 instruction, L1 data and L2 cache misses that affect the execution time of the application. In the multi-core configuration, the events of interest are the same, since they are the ones creating traffic in the bus, which is shared with the rest of the cores.
As we have explained in the beginning of this chapter, software randomisation changes the memory layout of the program, causing different cache layouts to occur per program executions. Each program execution has the same number of instructions that perform memory accesses, however the different cache layouts result in different conflicts between the memory objects and therefore the number of misses is changed. In particular, the number of misses have a random nature when software randomisation is used. As such, those results could be potentially processed by EVT in order to obtain the number of misses with a given exceedence probability level. Subsequently, this number of misses can be fed to the multi-core time composable model described in D3.8 to obtain composable pWCET estimations, by processing only results obtained in isolation.
In Figure 22 we show the inverse cumulative distribution for the instruction cache misses for one configuration of the CRAND benchmark in isolation, on top of the PikeOS Native Operating System.
Code randomisation is able to create a significant variation (5000) in the number of cache misses in the CRAND benchmark, transforming the number of accesses to a random distribution.
In Figure 23 we display the data cache misses distribution for the same benchmark, which exhibit a similar behaviour. We don't present L2 cache misses because the fact that it is partitioned but shared among the other cores complicates their accounting. In fact the hardware infrastructure provides a single counter for the L2 misses, which does not allow to distinguish which ones are generated by each core. Therefore, since operating system handlers can be executed in the other 3 cores, the exact number of misses initiated by the measured application cannot be identified.

However, since the number L1 instruction and data misses become a random distribution and the L2 is unified, the number of the L2 misses will be a random distribution, too.
To sum up, in this subsection we have shown evidence that the software randomisation makes the number of misses of the application to follow a random distribution, due to the different memory layouts which changes their way they interact and their conflicts. Therefore, the software randomisation indeed emulates the effect of a random placement cache, despite is difference in the granularity, which is now a the size of the object (function, stack) instead of the cache line.

3.2 P4080

The second platform in which DSR has been ported is the PowerPC-based P4080. Given that the porting on this platform started after the first iteration of DSR on the FPGA COTS had been completed, it received all the upgrades of the final version for the FPGA version explained in the previous section. However, due to the reasons explained in Section 2, it has only been integrated with the industrial operating systems which have been selected for the case studies, PikeOS Native for the railway case study and PikeOS APEX for the avionics one.

3.2.1 Porting details

The DSR solution on P4080 includes eager relocation, reduced dynamic memory consumption and the updated and fully validated random cache line selection.
From functional point of view, the P4080 DSR port is identical to the LEON3 FPGA one, with only difference the range of possible memory addresses to achieve mapping to any cache line set. In the LEON 3 port, the configuration selected for the final project evaluations for both hardware and software solutions, and benchmarks and case studies, did not feature a second level cache. For this reason, the range was selected equal to the cache way size of the LEON3 instruction cache, 4KB.
However, the P4080 features a three-level cache hierarchy, with the first two levels private to each of the cores, while the third level is shared among the cores and can be configured as a scratchpad. In the configuration selected for the evaluation, the last level cache has been reserved for use from the PikeOS operating system as scratchpad. Therefore, the application and the software randomisation can only use the first two private cache levels. The second level cache is unified with total 128KB divided in 8 ways, with each cache line holding 64 bytes. As a consequence each cache way contain 256 cache sets with total capacity of 16KB. For this reason, each allocation for function relocations is increased for an extra 16KB. Therefore the memory consumption in the P4080 is higher than in the LEON3 for the particular configurations selected projectwise for the evaluations. However, if another configuration is used, the implementation is fully configurable and can be adjusted to it. For example, if the end user decides to disable the L2, or configure L2 to contain only data, the memory consumption will be decreased accordingly to the instruction way cache size which is the same with the LEON3 (4KB).
Since in the P4080 we have to randomly map each memory object in the 256 cachelines of each L2 cache ways, we need to select 8 bits from the 32 bits generated from the random number generator. Similarly to the refined implementation of the LEON3, we use the upper 8 bits to be able to provide high quality random selection, as shown later in the Evaluation section.
In contrast to the LEON3 DSR port which has been supported in more than 5 different RTOSes, the P4080 variant has been ported to fewer RTOSes, according to the needs of the case studies. In particular, we provide an implementation based on the PikeOS Native personality, required by the railway case study, and the PikeOS APEX personality, required by the avionics case studies.

3.2.2 Evaluation

In this section we provide the evaluation for the DSR implementation on the P4080 platform. Similarly to the FPGA platform, we provide evidence regarding the effectiveness of software randomisation and we present evaluation results based only on benchmarks over the PikeOS Native variant. Results with case studies are presented in the corresponding WP4 deliverables.

3.2.2.1 Cache line Randomisation

In this subsection we evaluate the random qualities of the software random number generator used to randomly select the cache lines to which memory objects are mapped. In the P4080 we select among 256 cache lines, which is the size of the L2 cache. Note that the L1 cache has a smaller way size of 64 sets, therefore selecting a random cache line in the L2 effectively randomises placement in the L1, too, as explained in detail in the deliverable D2.7.
For the evaluation of the randomness in the cache line selection, we follow the same methodology introduced in 3.1.3.1. The only difference is that we use directly the 8 upper bits of the PRNG instead of further reducing them to 7.
The results of the tests can be found in the following tables and similarly to the LEON3 port, the minimum pass rate is exceeded, confirming the high quality of our PRNG generation strategy.

Name	Proportion
Frequency	100/100
BlockFrequency	99/100
CumulativeSums	100/100
CumulativeSums	100/100
Runs	99/100
LongestRun	99/100
Rank	100/100
FFT	96/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100

Name	Proportion
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100

Name	Proportion
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	97/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100

Name	Proportion
NonOverlappingTemplate	98/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	96/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	100/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	98/100
NonOverlappingTemplate	99/100
NonOverlappingTemplate	100/100
OverlappingTemplate	99/100
Universal	96/100
ApproximateEntropy	98/100
RandomExcursions	87/88
RandomExcursions	88/88
RandomExcursions	88/88
RandomExcursions	85/88
RandomExcursions	87/88
RandomExcursions	88/88
RandomExcursions	88/88
RandomExcursions	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	87/88
RandomExcursionsVariant	85/88
RandomExcursionsVariant	86/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88

Name	Proportion
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	88/88
RandomExcursionsVariant	87/88
Serial	99/100
Serial	100/100
LinearComplexity	100/100

Overall, we observe that the quality of the software random number generator 8 bit subset and therefore the cache line randomisation generated by its use in the software randomisation, is very high, similar in the case of the FPGA COTS, despite the difference in the number of the bits used in each platform.

3.2.2.2 Memory Overheads

For the evaluation of the memory overheads of DSR on P4080 we follow the same methodology with the one followed in the FPGA at Section 3.1.3.2.

Static Memory Overheads

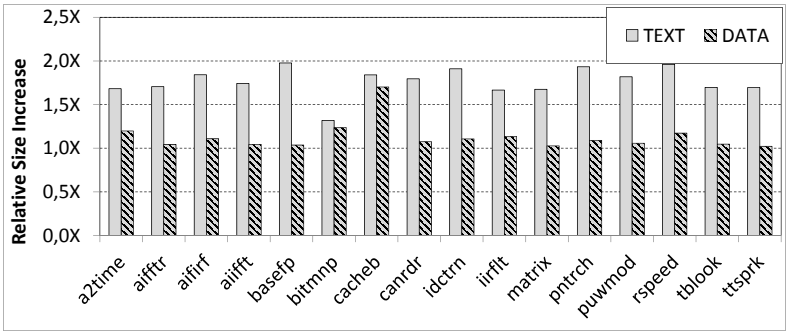

Figure 24: Memory overhead for different binary sections. Results are normalised to the binary generated without software randomisation.

For the evaluation of the static memory overheads created by DSR in the code and data segments, we use the EEMBC benchmark suite. Note however, that the P4080 integration has been completed late in the project and the board was time-shared between 3 partners (RPT, BSC, UPD). For this reason, although the integration of the benchmark suite with the RTOS has been performed, it was not be possible to occupy the board for long time in order to collect those results without compromising the development and experiments of the rest of the partners. Therefore we only use the EEMBC suite to present data collected statically by compiling the benchmarks to generate executables linked with the DSR library in this section, but we don't provide execution time results with them in the following sections.
Figure 24 shows the memory overheads for both code and data binary segments of the EEMBC benchmarks. Due to the fact that the P4080 has a different ISA from FPGA, we observe that the memory overheads are slightly higher in the P4080.
In particular, the code size is almost doubled in P4080 while in FPGA this increase was 50%. The main reason for this increase is the impact of the PowerPC ISA in the code generation for the stack frame manipulation. PowerPC doesn't feature a register window as the FPGA which is based on SPARC. Therefore a significant amount of code

Table 7: Worst-Case Stack Usage in various SRAND configurations due to stack randomisation, with total code size 125% of the L2 cache

Max Depth	Original Stack (Bytes)	Randomised Stack (Bytes)	Relative increase
6	738	50658	68,64×
26	3198	219518	68,64×
51	6396	439036	68,64×

is generated in each function in order to save and restore registers between function invocations. However, similar to the case of the FPGA this increase is not actually so big, but its relative increase is high due to the small size of the EEMBC benchmarks compared to size of the software randomisation library.
Regarding the data segment, the increase is also higher in P4080, 13% on average. The reason is again related to the ISA. More concretely, in P4080, floating point numbers need to have a 64-bit alignment in order to avoid a performance penalty, so the linker respects that alignment in expense of more occupied memory.
In both cases, we see that the static overheads of software randomisation are not significant in P4080, especially when applied in industrial size code, which makes their relative contribution negligible.

Dynamic Memory Overheads

For the same reasons with the FPGA DSR evaluation, we have used the CRAND and SRAND benchmarks to evaluate the worst case stack and heap usage with DSR.
In Table 7 we can see the worst case stack consumption for various SRAND configurations. Similarly to the PROXIMA FPGA board, the overhead over the non-randomised version is constant, since it depends only on the function size in the binary and not on the maximum stack depth. Also we notice that the overhead is higher in the P4080, due to the fact that we use larger padding, equal to the L2 cache way. This number is 4 times bigger than the instruction way cache size used in the FPGA, and for this reason the overhead is almost 4 times larger. As explained in the introduction of the P4080 section, this overhead depends on the particular configuration that the hardware is used. If the user selects to disable the L2 cache, the overhead of software randomisation will be reduced, as the padding will need to match the L1 way cache size.
The relative heap consumption on the other hand is slightly lower in the P4080 than in the FPGA compared to the results presented in the FPGA section. Note that again we have used the CRAND configuration which has total code size equal to 125% of the L2 cache size. In the FPGA section, we used a hardware configuration without L2, therefore the size of the benchmark was configured relative to the L1 instruction cache. In order to have meaningful evaluation scenario for the evaluation of software randomisation we adopted the size of the benchmark to that size.
Recall that the necessary padding that we have to add in order to be able to map functions in any cache line of the instruction cache is equal to the L2 cache way size, which is 16KB. Therefore it is higher than in the FPGA in absolute terms, which it was 4KB. However, due to the fact that the original (non-randomised) size of the benchmark is now significantly larger (relative to the L2 cache size), the relative overhead is reduced.
Overall, we see that software randomisation introduces some non-negligible memory overheads. However, those memory overheads are possible to be bounded, are in acceptable ranges relatively to the RAM sizes used in current and future industrial systems, and their impact is mitigated in industrial size software, since their relative increase is smaller

Table 8: Worst-Case Heap Usage in various CRAND configurations due to dynamic code allocations

# of Functions	Total Code Size (Bytes)	Total Allocated Size	Relative increase
4	156024	237944	1,52×
8	155900	303644	1,95×
16	157175	435737	2,77×

in that case. Finally, the benefits from simplifying the analysis due to the MBPTA that is now possible with software randomisation on conventional architectures, overcomes the potential impact of those overheads.

3.2.2.3 pWCET Estimations

For the reasons we explained in the previous subsection, we do not use the EEMBC benchmark suite for the pWCET estimations. Instead we are using our CRAND and SRAND micro-benchmarks, in addition to the railway and avionics case studies which are presented in D4.8.

Code size smaller than the cache

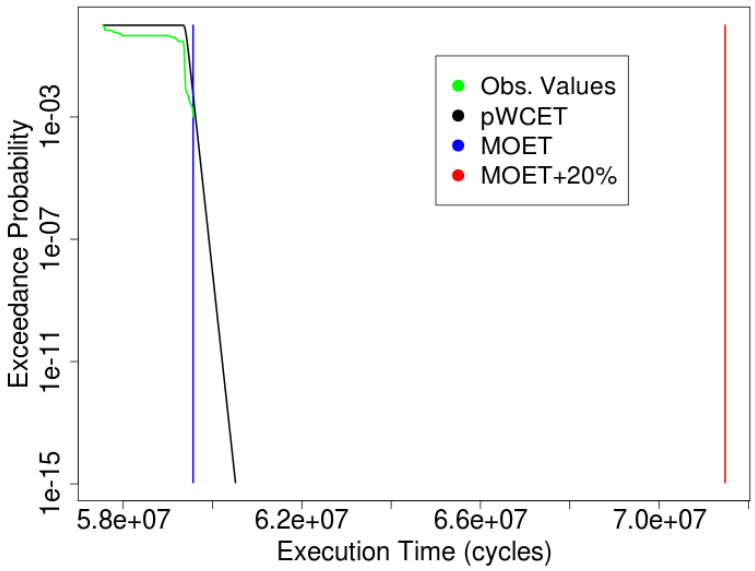

Figure 25: pWCET estimation for CRAND with 4 functions configured to use 75% of the L2 cache.

Figure 25 the pWCET estimation for CRAND using 4 functions configured to use 75% of the L2 cache, while Figure 26 presents execution time of the same configuration of the benchmark without software randomisation. First we notice that software randomisation creates significant variability in the execution time (Figure 25 in green), as opposed to non-randomised configuration which experiences minimum variability (Figure 26 with green) due to the fact that the code fits in the L2 cache. This is very important since it shows that software randomisation is effective even in this "unfriendly" configuration, which is not expected to perform optimally as explained in D2.7. Moreover, the two-level private memory hierarchy of P4080 in combination with its relatively large caches, makes this scenario frequent with several applications executed on this platform.

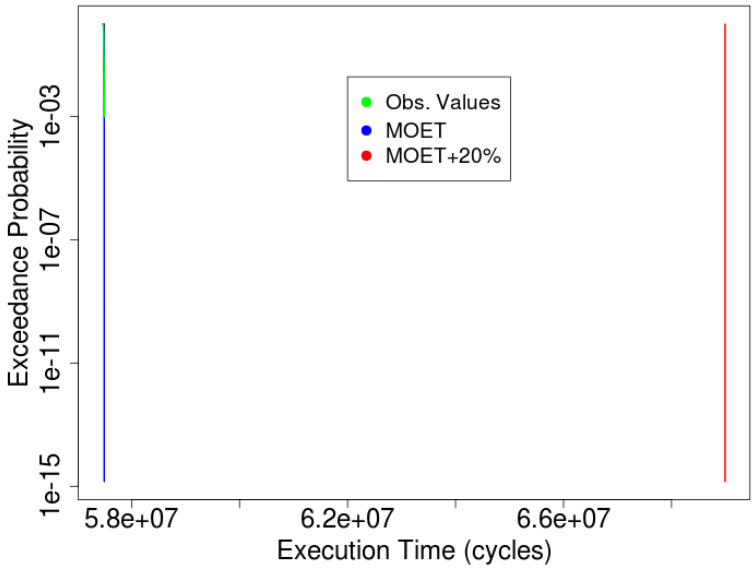

Figure 26: Observed execution time and WCET with current industrial practice for the same configuration of CRAND, without software randomisation.

Regarding average performance, we notice that the software randomised version is slightly slower (1%) than the non-randomised one. However, this degradation in average performance is diminished when considering the worst case performance, thanks to the benefits obtained by enabling the used of MBPTA. In the non-randomised version of the benchmark, the WCET is estimated using the current industrial practice of adding an engineering margin of 20% over the maximum observed execution time (MOET). However, the pWCET estimation obtained by MBPTA (Fig. 25 in black) for exceedance probability level of $10^{-}15$ is 14% lower than the corresponding WCET in the non-randomised case (Fig. 26 in black).
MBPTA results are not only competitive with the current industrial practice when considering the non-randomised case, but also with the software randomised solution. In fact the pWCET estimation obtained by MBPTA (Fig. 25 in black) for exceedance probability level of $10^{-}15$ is 18% lower than the WCET obtained with a 20% engineering margin (same figure in red).
In Figures 27 and 28 we see the corresponding results for the CRAND benchmark with total code size 75% of the L2 cache, but with 16 functions. Similarly the randomised version exhibits significant variability, much more than the 4 function configuration. The reason is that in the randomised version in some memory layouts the code of the functions is overlapping creating more conflicts. This is not the case in the non-randomised version since the code of each function is placed in consequtive memory locations and thus it always fits in the cache, creating negligible variability from run to run.
In terms of average performance, the software randomised version exhibits a 2% slowdown, while in WCET performance we have a small difference from the 4 function case. When comparing the WCET estimations for the randomised and the non-randomised cases, the MBPTA gives a slightly tighter result than the current industrial practice (less than 0.1%) for exceedance probability 10^{-12} than the industrial practice, while for probability 10^{-15} is 3% slower. Therefore in this case, the estimates obtained by the two techniques are equivalent. However, the software randomisation provides two additional benefits. First allows the used to select a configurable WCET depending on the critical-

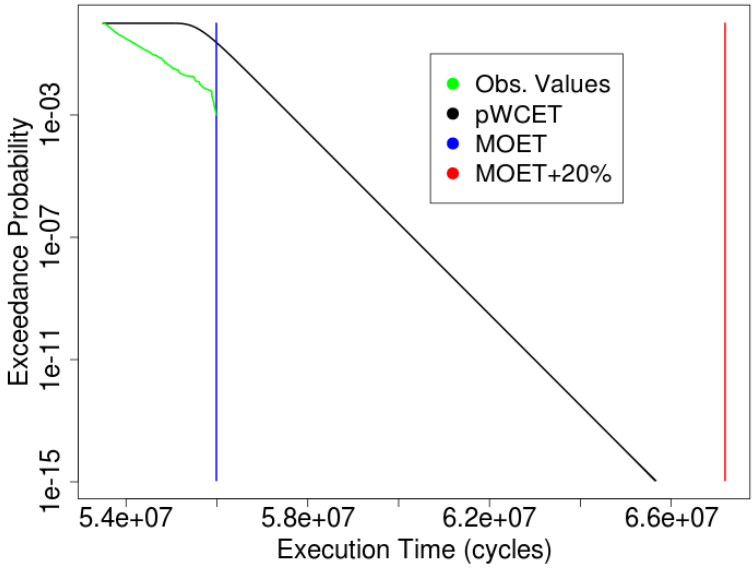

Figure 27: pWCET estimation for CRAND with 16 functions configured to use 75% of the L2 cache.

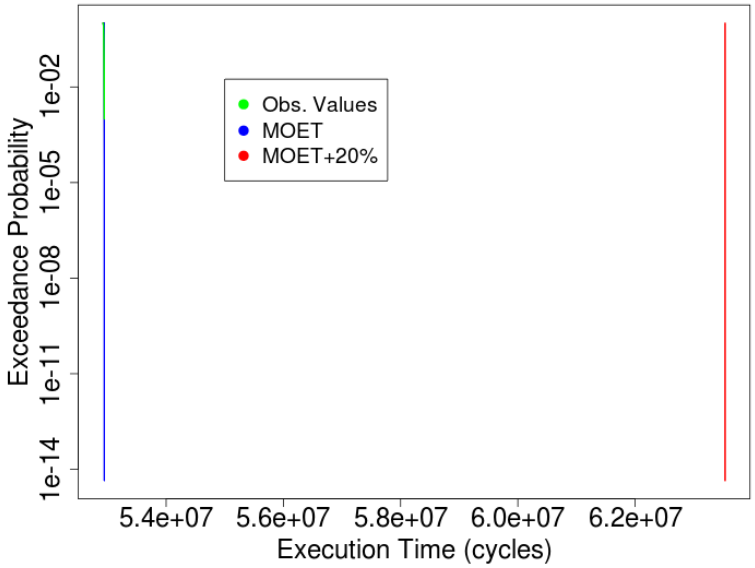

Figure 28: Observed execution time and WCET with current industrial practice for the same configuration of CRAND, without software randomisation.

ity of the task, instead of a constant uppperbound. For example, for a lower criticality task, an exceedance probability lower than 10^{-12}, eg. 10^{-9} can be selected, allowing for greater WCET reduction. The second benefit, whose importance is even more significant, is that software randomisation allows the industrial user to take into account the effect of the cache indused source of jitter due to the memory layout in the WCET computation, therefore obtaining robust WCET results in the presence of composition. In particular, linking the application with other libraries or the RTOS at the final integration phase changes the memory layout of the program. Therefore, in the non-randomised case, even

if this additional code linked with the application is not executed during the measured part of the application, this change in the memory layout can have a significant impact on the execution time in the integrated binary, possibly different from the one observed in isolation during the development phase, complicating the incremental development process followed in industry. On the other hand, this integration is not going to have any impact in the software randomised application (assuming that the linked code is not invoked during the measured part of the application), since the different memory layouts have already been observed during the measurement collection phase.
Finally, when comparing the current industrial practice and MBPTA on the randomised application, MBPTA is 5% tighter for 10^{-12} cut-off probability and 2% tighter for 10^{-15} probability.

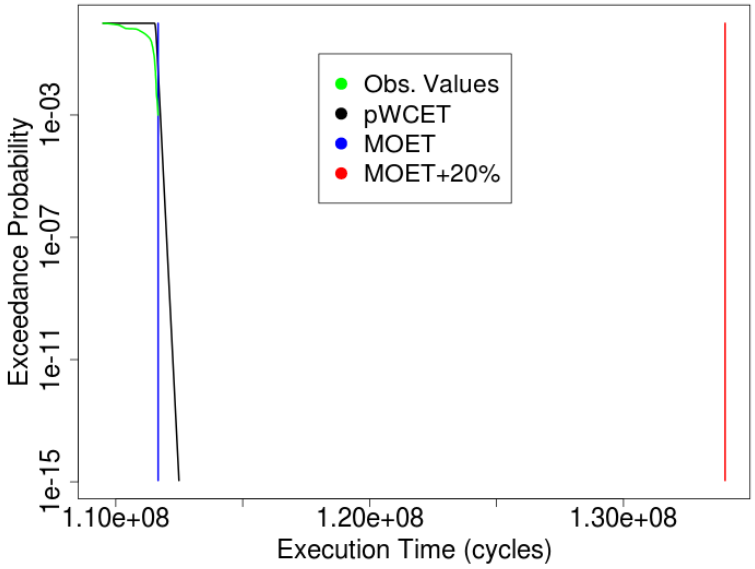

Figure 29: pWCET estimation for CRAND with 4 functions configured to use 125% of the L2 cache.

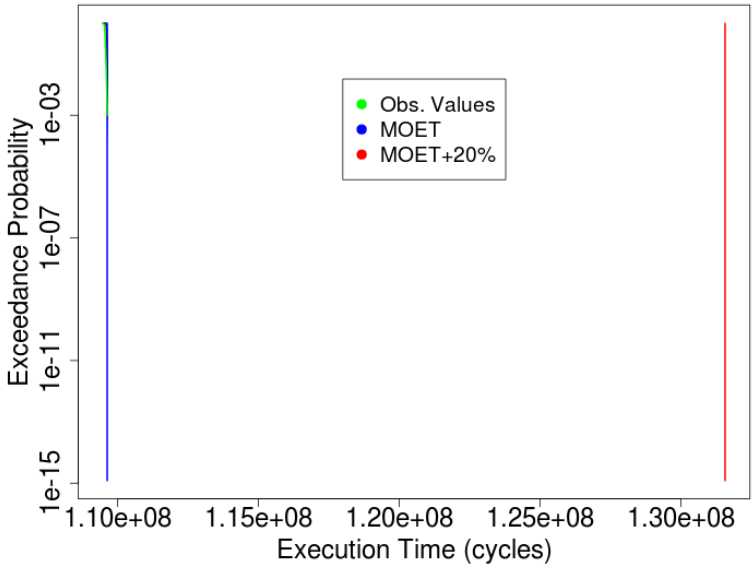

Figure 30: Observed execution time and WCET with current industrial practice for the same configuration of CRAND, without software randomisation.

Code size bigger than the cache

Similar results are observed when the size of the code exceeds the capacity of the L2 cache, as shown in the results presented in Figures 29 and 30. Again the software randomised version exhibits significant variability in the execution time, in constrast to the non-randomised version. Note that in sets of possible execution times explored by software randomisation, there are few ones faster than the non-randomised version, even with the overheads of software randomisation. The reason is that software randomisation is able to explore some memory layouts that have better cache utilisation than the default

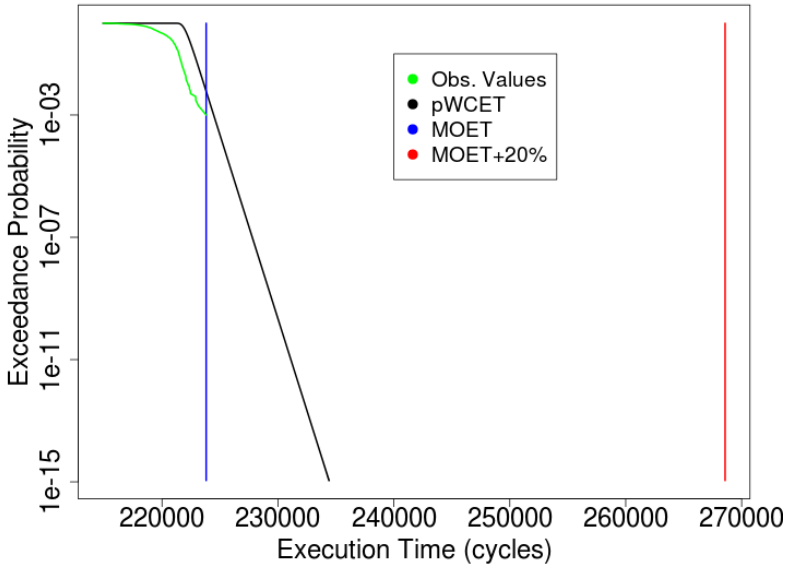

Figure 31: pWCET estimation for SRAND with maximum stack depth 6.

memory layout when no randomisation is used.
The degradation in the average performance due to randomisation overheads is minimal, as in the previous case, less than 1%. Regarding the WCET comparison between the randomised case with MBPTA for cut-off probability 10^{-15} and the non-randomised with current industrial practice, again the MBPTA provides a tighter estimation by 16%. Focusing on the software randomised version of the benchmark, MBPTA for the same probability level is 18% tigher than the current industrial practice.

Stack randomisation
In Figures 31 and 32 we show execution time results for the SRAND configuration that has maximum stack depth 6. As opposed to the code randomisation benchmark, both the randomised and non-randomised programs exhibit significant variability in the execution time. However, as expected, the randomised version exhibit the double amount of variability. In particular, the randomised version has a variability of 10K cycles, which is 5% of its execution time, while the non-randomised version 5K cycles which is 3% of its execution time.
The average performance of the randomised version is 24% larger than in the non-randomised one. The reason for this is that both code and stack randomisation are applied to the program, so the degradation is due to the overhead of both. Moreover, since the length of the program is small, the increase in relative terms is significant.
Due to this average performance degradation, the current industrial practice on the non-randomised application provides a tighter WCET estimation than the MBPTA (9% for probability 10^{-15} and 8% for probability 10^{-12}). However, the benefit of the software randomisation is that this WCET is robust in memory layout changes that can occur in incremental development. Any change in the memory layout of the non-randomised application, can cause a severe execution time degradation.
Comparing MBPTA and current industrial practice on the randomised application, results in MBPTA providing 14% tigher WCET for 10^{-15} exceedance probability and 16% for 10^{-15}.
In general, in this subsection we have seen that the MBPTA which is enabled by soft-

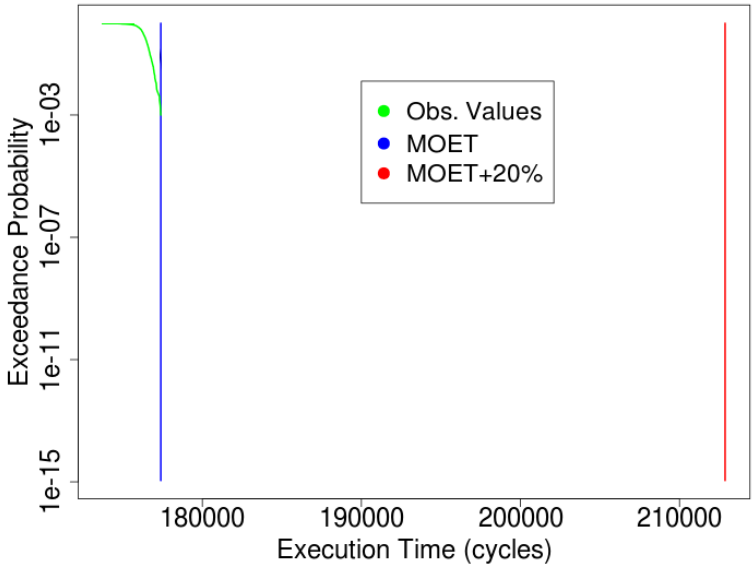

Figure 32: Observed execution time and WCET with current industrial practice for the same configuration of SRAND, without software randomisation.

ware randomisation on the P4080, provides competitive results compared to the current industrial practice, not only in the software randomised version of the application but also when compared to the non-software randomised one. However, in the latter case, MBPTA provides higher confidence on the WCET estimate, because the obtained number takes into account the variability caused by the memory layout and this estimate is also preserved in the case of integration.

3.2.2.4 Event distributions

Similarly to the FPGA case, the miss distributions in P4080 obtain a random behaviour due to software randomisation.
In Figure 33 we see the instruction cache miss ICDF distribution when the code fits in the L2 cache for various single and multicore configurations, using a CRAND configuration with 4 functions. In the multicore configuration, the application is together with a contending application mapped to another core that misses constantly in its L2 cache. First we observe that for both randomised and non-randomised applications the number of instruction misses follows the same distribution regardless of whether it is executed in isolation (single-core in the graph legend) or in against a contending application. Of course, small descrepancies are expected due to the operating system noise. The reason that the miss distributions are not affected by the presence of contention is that in P4080 the first two levels of the cache are private per core, while in the current configuration, the shared L3 cache is only used by the OS. Therefore, the contenting task has no effect on the cache contents of the task under analysis.
The non-randomised version of the application (named Static in the Figure) exhibits lower number of misses since the code fits in the cache, and has almost no variability as expected. The reason for this is that the code fits in the cache and also the memory layout remains always fixed.
On the other hand, in the randomised version the miss distribution has a random nature, as it is shown by the large variability of its ICDF distribution. Recall that due to this

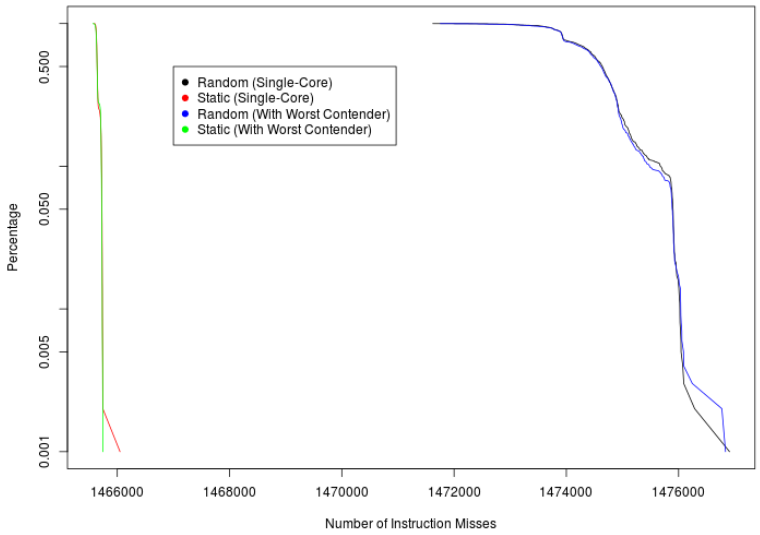

Figure 33: Instruction cache misses for CRAND with 4 functions and code size 75% of the L2 cache

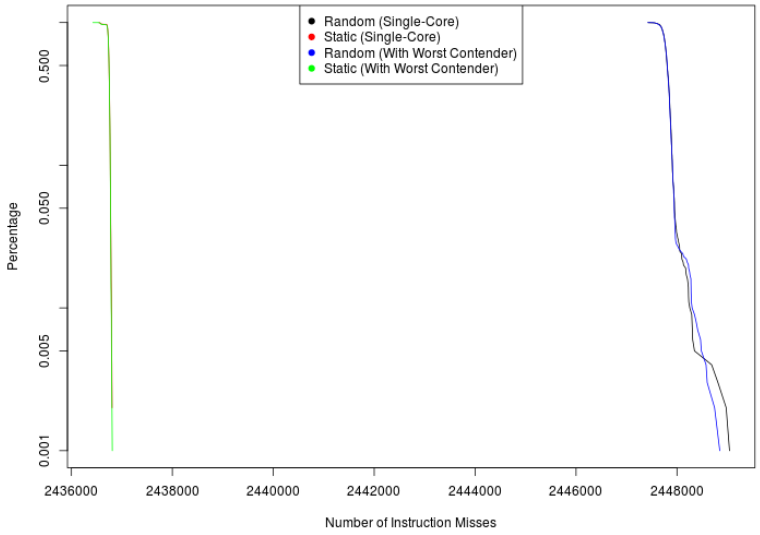

Figure 34: Instruction cache misses for CRAND with 4 functions and code size 125% of the L2 cache

random nature, this number can be fed to the multicore time composable model described in D3.8 to obtain time composable pWCET estimations.
In Figure 34 we can see the same distributions for the case that the code exceeds the L2 cache. In this case we can observe that the variability in the cache misses of the software randomised task is smaller than in the case that theh code fit in the cache. This happens because the cache miss number is dominated by capacity misses, therefore cannot vary much regardless of the memory layout.
The exactly same observations can be made for CRAND when using 16 functions, as shown in Figures 35 and 36, as well for the data cache miss distributions which are

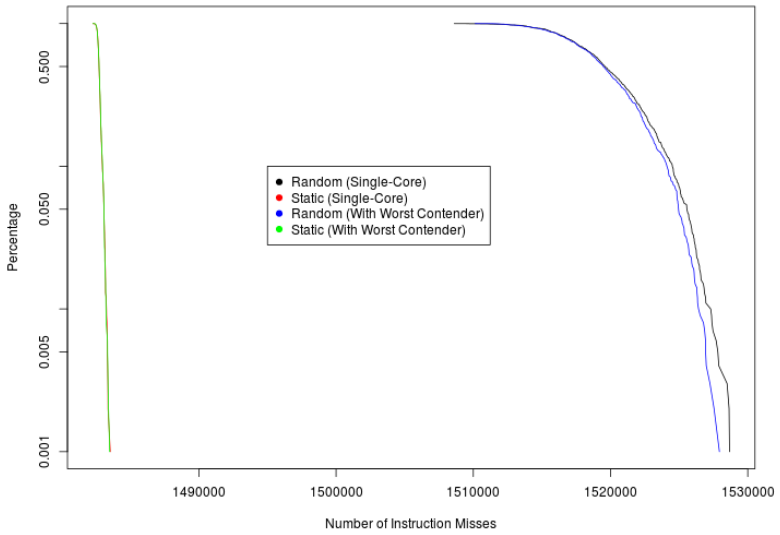

Figure 35: Instruction cache misses for CRAND with 16 functions and code size 75% of the L2 cache

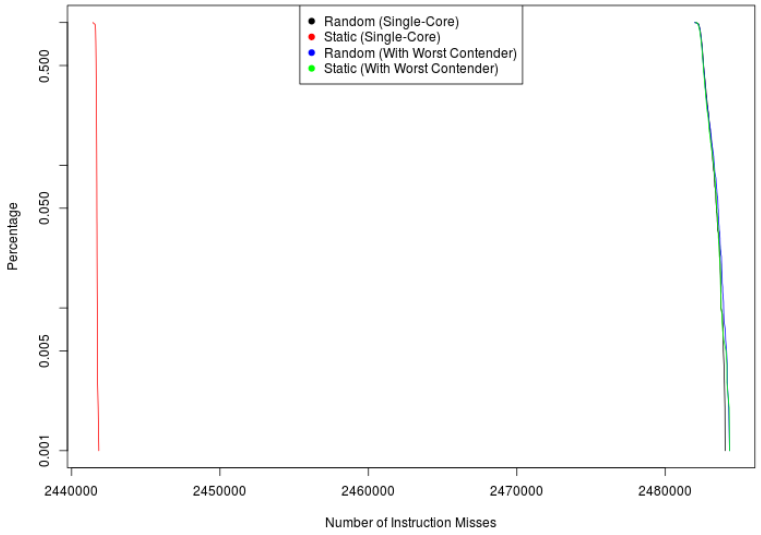

Figure 36: Instruction cache misses for CRAND with 16 functions and code size 125% of the L2 cache

presented in Figure 37.
Therefore in this subsection, we have shown experimentally that adding a random amount of padding equal to the L2 cache way size, not only randomises the L2 cache placement, but also the L1.

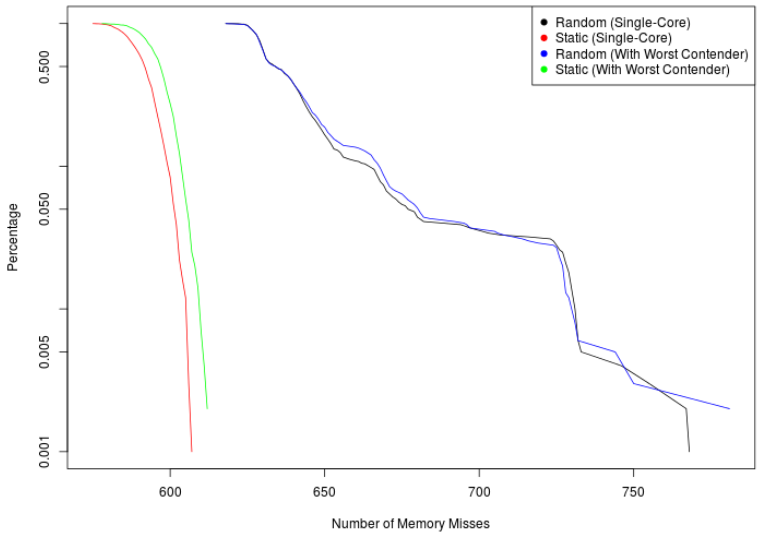

Figure 37: Data cache misses for SRAND with maximum stack depth of 6.

3.3 AURIX

While the software randomisation in the FPGA and the P4080 platforms is based on DSR, this was not possible in AURIX. The reason is that this target platform, whose architecture is shown in Figure 38, features a read-only Flash memory device, which contains the code and read-only data of the software. For this reason, the self modified code that is required for DSR in order to implement code randomisation is not possible to be used.
Moreover, the LLVM compiler that DSR is based on, does not have an AURIX backend. To make matters worse, AURIX has a binary only industrial compiler toolchain based on GCC. Although the source code for an old version (last updated in 2008) of the compiler is available from [7] as required by the GPL licence of gcc, that version is not compatible with the new boards that are available in the market, like the ones used in the project.
For these reasons, in order to provide MBPTA compatibility on AURIX, we have developed a static variant of software randomisation. In order to overcome the challenge of being unable to apply our changes in an AURIX compiler, we designed this variant to work at the source code level of the application. This way it becomes agnostic of the particular industrial toolchain, and for this reason we named it Toolchain-Agnostic Software rAndomisation (TASA) [11].
The fact that this method is not tied to a specific compiler can facilitate the adoption of software randomisation in industry.
Additionally, TASA reduces the tool development cost and most importantly the tool qualification cost required for industrial toolchains which are used in critical real-time domains. Since TASA is an independent tool, there is no modification and therefore requalification need and cost for the industrial compiler provider for each of the supported platforms. Instead TASA needs to be qualified once, minimising this cost.
It is important to note that TASA was not included in the initial roadmap of the project. The provisioned solution and work estimation for software randomisation on AURIX was much lower than the effort required to build a software randomisation solution from scratch. Instead TASA emerged as a need to solve the MBPTA compliance of a platform that would not be possible otherwise and a significant amount of unplanned effort has been devoted for its development. This fact in addition to the fragile components of the AURIX toolchain, the lack of detailed documentation or support by the vendor and the short time remained after several refinement iterations between the AURIX involved partners did not allow TASA to reach as high maturity level as DSR. Despite that, TASA has been successfully evaluated for the purposes of the project.
In the following we examine the improvements of TASA since the previous reporting period, the challenges we faced and how they were overcome, its current limitations and finally its evaluation.

3.3.1 High Level Description of Static Software Randomisation/TASA at m18

As explained in the introduction of this Chapter, software randomisation emulates the work of a hardware random placement cache. Since the modification of the cache mapping is impossible, the equivalent effect of mapping memory objects in different cache lines is achieved in an indirect manner, by modifying the position of memory objects in the main memory across different executions of the program. In Dynamic Software Randomisation this is achieved by moving memory objects during the program execution. However in AURIX this is not possible due to the presence of read-only flash memories, which prevents this process.
Static Software Randomisation achieves the equivalent effect by generating multiple ex-

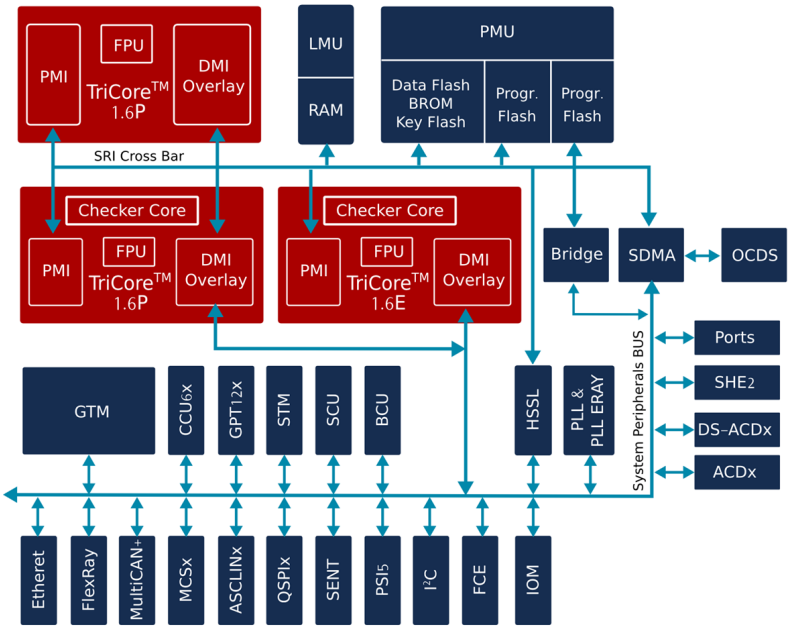

Figure 38: Architecture of the AURIX platform

ecutables with different random binary layouts. Before the program is executed, the binary is loaded in the memory, and the memory objects are placed randomly in different memory positions.

In order to randomise the binary layout, TASA is based on the principle that compilers generate program elements in the order they are encountered in the source file. This principle is only violated by a very small number of compiler options that we identify and disable, without any impact in code size or execution time as our results indicate.

TASA is implemented as a standalone source-source-compiler for ANSI C, which randomly shuffles program elements: functions, local, global variables, struct definitions. This results in code randomisation, stack randomisation and global/static variable randomisation.

Code Randomisation is further enhanced with the introduction of a random amount of non-functional, non-executed code in the form of assembly level *nop* instructions. This way, the achieved Code Randomisation is equivalent to the DSR implementation.

Stack randomisation is achieved in two orthogonal ways: intra-stack randomisation, i.e. randomisation of memory objects in the data cache coming from the same stack is achieved by shuffling their definitions, while inter-stack randomisation, that is randomisation of memory objects between different function calls is achieved by introducing a random padding in the stack frame. The latter solution is equivalent to the stack frame randomisation implemented in DSR, and in the case of TASA is achieved by the introduction of non-used local arrays.

The implementation of TASA at m18 was only restricted to the ANSI subset of C and it had very limited preprocessor support. The reason for this was that the support of subsequent C versions or compiler vendor extensions would increase significantly the complexity and the implementation effort for a custom C parser.

On the other hand, the implementation of TASA in a compiler front-end such as gcc or clang (LLVM's frontend) would solve the parsing problem, but it would increase the complexity and implementation effort to unaffordable levels for the remaining period of the project, given that the project consortium didn't have the required expertise and those tools have a very steep learning curve.

3.3.2 Limitations of TASA at m18 and their solutions

After the initial TASA implementation has been released and tested by the project partners involved in the AURIX the following limitations have been identified and solved:

3.3.2.1 *typedef* and dependent declarations

One of the limitations of TASA is that it does not keep track of *typedef* and other dependent declarations while it shuffles definitions. The reason for this is that TASA does not implement a symbol table, since this introduces significant implementation complexity that would prevent TASA to be ready and validated early to be used by the case study application before the end of the project. However, this can potentially result in code that does not compile, if the random emitted order uses first the declaration of a variable, without being preceded by its type definition.
To overcome this problem we use two methods.

- Type definitions are typically defined in header files and included by other files. Therefore, we apply TASA only on the source files. Since TASA does not implement the preprocessor functionality, to keep the implementation complexity low, it doesn't parses the contents of the included files and therefore the type definition cannot be placed after a variable declaration. In the case that the type definition is in the same source file, the code is refactored by placing the elements that should not be shuffled in a separate header file.

- There are some case though, that this strategy cannot be applied. In particular, there was a strong requirement generated by the WP3 consortium members (RPT and UoY) to apply TASA over the instrumented version of the source file, in order to guarantee the consistency of the instrumentation points identifiers between the software randomised versions of the source code. Since the instrumentation implementation of Rapita is based on the preprocessor, this means that TASA has to process the contents of the header files, too, which are included by the processor.

  To deal with this problem, special support was added to TASA in order to ignore specific portions of a source file. The end user, is responsible to identify those parts, eg. the type definitions, and annotate their beginning and end manually with preprocessor *pragma* directives. Alternatively, header file preprocessor includes can be annotated in the same way, achieving the same effect with the previous solution, that is to not shuffle their entire contents.

3.3.2.2 Vendor-Specific/Non-ANSI Extensions

With the integration of the CONCERTO case study automotive application on top of the ERIKA-TC RTOS explained in the AURIX section of the previous Chapter, the requirement from WP3 regarding the parsing of preprocessed files pushed TASA's parsing capabilities to its limits.
The application need to include synchronisation primitives and other low-level operating system constructs, increased the number of new elements that needed to be supported by TASA. This included a lot of improvements in order to support various keywords such as *inline*, parsing of assembly blocks and more importantly special attributes that are used abundantly in AURIX to specify placement of individual memory objects in the different memory devices (flash, scratchpad, cache, LMU etc)
Although successful, the implementation of these non-standard features cannot be considered complete. Instead it is limited to the constructs required for the compilation of the CONCERTO case study over the ERIKA-TC. This means that there might be

specific versions of the supported non-standard extensions, or additional non-ANSI or vendor-specific extensions in other case studies/RTOSes that TASA is not able to parse correctly. For this reason, we consider that TASA is not yet in the same maturity version as DSR.

3.3.2.3 preTASA

Some of the proposed solutions in the above problems required the manual input from the user in the form of annotations. However, this solution is not appropriate for industrial use of TASA, and could potentially create problems in the evaluation of TASA by IAB members. For this reason, Rapita has provided a tool called preTASA, that provides a seemless integration of RVS with preTASA.
preTASA parses the preprocessed code before it is fed to TASA. preTASA introduces automatically annotations around header files, allowing TASA to be able to parse only the part of the source file that actually affects the memory layout (function implementations, variables' declarations), without having to deal with additional complexities and unsupported constructs.
preTASA is based on Rapita's instrumenter parser and for this reason is provided only in a binary form for Windows and Linux platforms. Despite Rapita has no plans over future improvement and maintenance of preTASA, the tool comes with a scripting feature in C, which can be used in order to add new functionality to it.

3.3.3 Evaluation

In this section we provide the evaluation of TASA using benchmarks. TASA results obtained with the CONCERTO case study, are provided in the D4.8. Moreover, the complete multi-core analysis is provided in D2.8.
In the previous reporting period, a very limited evaluation was presented in D2.7, due to the fact that TASA was in its early stages of development. In the subsequent interval, significant efforts had been devoted towards the improvement of TASA and its successful integration with the CONCERTO case study and the research oriented RTOS ERIKA-TC. In fact the integration process has been proved very challenging due to the very fragile framework composed by the vendor provided compiler toolchains, RTOS, board versions and tracing subsystem.
Initially, 3 boards were provided by Infineon and several development kits had been purchased by project partners in order to be able to advance each part independently (RTOS, TASA, tracing, multicore interference) without time consuming dependencies. After long periods of constant and many times unsuccessful attempts to solve various problems caused by the composition of the framework's elements, we have realised that there were extremely close dependences between particular SDK versions and board versions.
In particular, each part of the toolchain developed by different partners was able to work in isolation with the SDK version and specific board of that partner, however it was not possible to reproduce the correct functionality in some other boards owned by the same partner or others.
However, neither the software nor the hardware elements were able to be selected by each partner. Only the latest SDK has been available by Hightec, without the possibility to downgrade to the required version known to work in other partners configurations. Similarly the hardware vendor did not offer the possibility to buy the specific board supported, since they were out of stock and only newer but unsupported batches were available.
In order to be able to have the integration ready and finish the evaluation with the

case study on time, we have decided to identify the subset of the boards/SDKs that work properly with the integration of all tools and use them for the case study. Out of the 6 boards, only 2 have been identified to properly working with the SDK/RTOS/-TASA/tracing combination. For this reason, these boards have been time-shared between partners, until one of them became unavailable, due to the fact that was provided to an IAB member for a case study evaluation.
As a consequence, the problems we faced for long time for the toolchain integration, together with the small availability of the board due to the case study evaluation and time-sharing between the partners, did not allow us to provide a complete evaluation as we intended. However, we were able to collect sufficient evidence that software randomisation is effective on a completely deterministic platform using the case study application, in addition to the micro-benchmark results presented in D2.7 in the last reporting period.
Moreover, since the AURIX board was not fully available for evaluation, we have evaluated TASA on the FPGA. Although this is a different platform, the principle of TASA is the same and TASA itself is completely platform independent, since it is implemented in the source code level. Moreover, this provides the additional benefit of being able to compare DSR with TASA and evaluate better its overheads and MBPTA qualities. Finally, it demonstrates that TASA is completely portable to any platform.

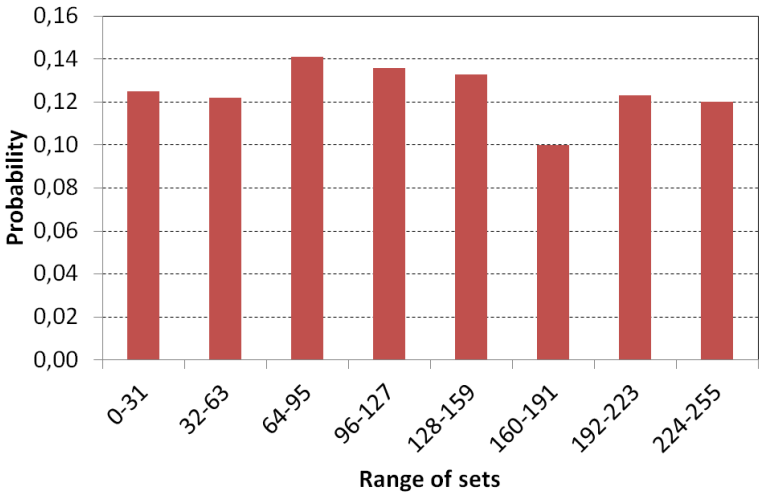

Figure 39: figure
Cache Set distribution of the first instruction of FUNC1 in set groups.

3.3.4 Cache line Randomisation

In DSR, the memory allocator always returns page aligned entries, which are mapped in the first set of the cache way. Then this address is then padded by a random number generated by a software PRNG up to the cache way size to randomly select a random cache line.
For the FPGA and P4080 we have evaluated the randomness of the PRNG bits used to select the cache line.
On the other hand, in TASA the position in the cache of each function depends on the previous functions. That is even in the case that no padding is applied, the placement of each function is essentially random. Since the number of possible function ordering might not be sufficiently random, we also add a random padding, similar to the DSR. However due to the dependency on the previous functions and the particular characteristics of each application (number and size of each function), this randomness cannot be quantified

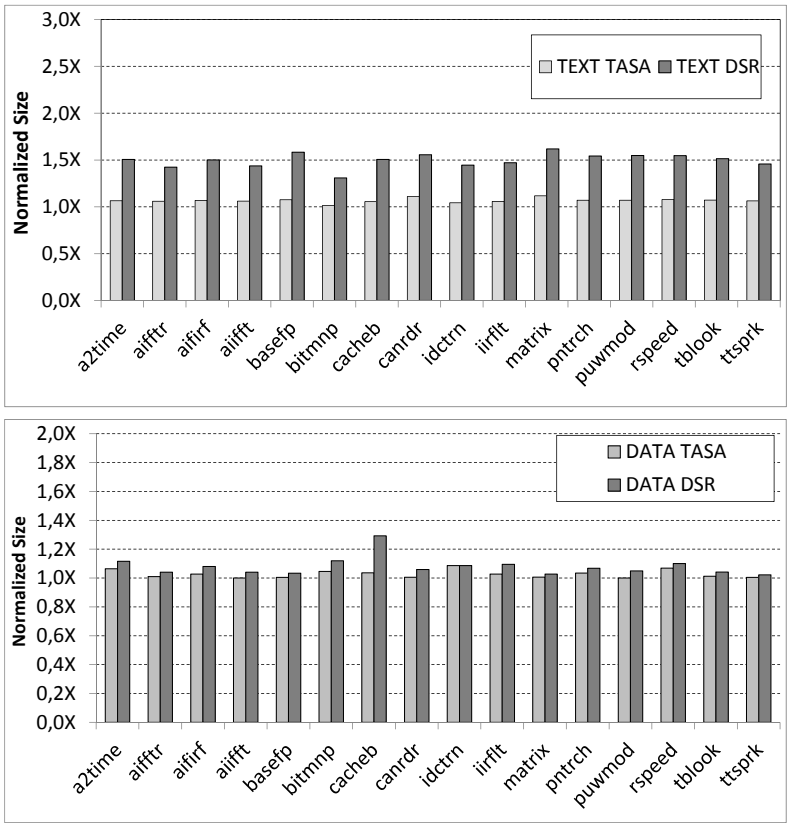

Figure 40: Memory overhead for different binary sections. Results normalised to the binary section sizes without software randomisation. Values for TASA are average for all binaries.

using the tests we used for DSR.
As an alternative way to evaluate the cache line randomisation, we have arbitrarily selected a function of the cache study, and we observed the cache line in which it has been placed, by analysing its position in the 1000 generated binaries using TASA.
As it can be observed in Figure 39, TASA achieves a fairly balanced distribution across binaries and therefore program executions. Note that for a higher number of binaries, the probability of each group converges to 12.5%, due to the uniform random nature of TASA. This proves that TASA randomises effectively the cache memory layout.

3.3.5 Memory Overheads

For the evaluation of the memory overheads of TASA we follow the same approach as with the previous two platforms with DSR, dividing the overheads to static and dynamic ones.

Static Memory Overheads

For the reasons we explained in the beginning of the evaluation section, we used the EEMBC benchmarks compiled with TASA but for LEON3. Although the AURIX ISA is different from the SPARCv8, both of them are RISC-like and therefore the results follow the same trend. Moreover the fact that we compile them for LEON3 allows us to compare TASA with DSR, since it is available for this platform.
Figure 40 shows the memory consumption increase for the `.text`, `.data` segments for the EEMBC Automotive when using TASA and compares it with the corresponding increase

with DSR. Each software-randomised setup is normalised to the same non-randomised configuration, e.g. TEXT TASA provides the relative increase in the text segment when TASA is applied over the non-randomised one.

Code Segment.

Code padding increases the application's code footprint when TASA is used, since it is introduced in statically in the binary, in the form of a random amount of never executed nop instructions, up to the instruction cache way size. This overhead is less than 7% on average and is related only to the memory footprint, since it is never executed.
In contrast, in DSR this overhead is higher, close to 50%. As we explained in the DSR evaluation, this comes mainly due to the big size of the DSR runtime which is linked with the application, and its effect is exacerbated from the small binary size of the EEMBC benchmarks. Note that a small amount of this 50% added code in DSR is responsible for stack randomisation. This code is actually being executed so it increases also the executed instruction count. On the other hand, no additional code is generated for the stack manipulation in the TASA.
Another difference between DSR and TASA, is that in DSR the memory overhead for the code randomisation which is incurred in order to be able to place memory objects in any instruction cache set is not is not considered as a static overhead in the code, but as the dynamic memory overhead in the heap, while for TASA this overhead is linked to the *.text* segment size. Also, regarding its size, in DSR a padding of a cache way size is always added in the function allocation size. On the other hand, TASA allocates only the additional number of space which is decided during compilation time. This means that on average the overhead of TASA due to the padding is smaller than in DSR. However, in real-time systems is the worst case scenario that is important. This for TASA is the very unlikely situation in which a padding up to a way size is added in all the functions and it is exactly the same with DSR.

Data Segments.

For the `.data` section, the overhead of TASA is less than 7%(Figure 40). This overhead comes from the alignment of large global variables used in the EEMBC benchmarks which cause an increase in their size, especially large initialised arrays which are used to simulate input and output buffers.
On DSR this overhead is bit higher, in the range of 10% on averages and comes mainly by additional global/static variables actually included in the DSR runtime code.
Putting it all together. Overall, TASA incurs lower static memory overheads than DSR, causing only a 7% increase on the static memory footprint of EEMBC Automotive benchmarks.

Dynamic Memory Overheads

For the dynamic memory overheads we have stack and heap. TASA performs randomisation statically, and for this reason doesn't have an overhead in the dynamic memory allocated in heap, opposed to DSR.
Regarding stack, TASA increases the space used per function around 25% on average, due to the random padding up to the cache way size which added in order to be able to map a stack frame in any data cache set. On the other hand, the intra stack randomisation when local variables randomisation is enabled creates a variability of the stack size of $\pm 1\%$.
However, as we explained in the evaluation section of DSR on FPGA, in real-time systems

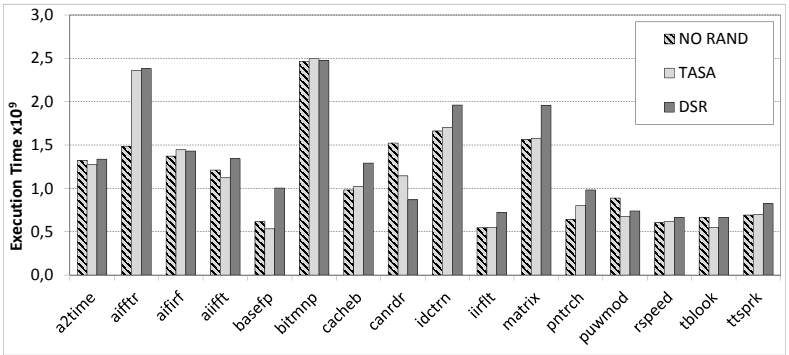

Figure 41: Average execution time measured in processor cycles for TASA and DSR.

is important the worst case stack consumption. Similarly to the code padding case which we examined earlier for the code padding overhead, the worst case scenario is that each function stack frame is padded with a data cache way size. TASA's stack consumption is identical to the DSR's one.
Overall we have seen that the worst case memory consumption of TASA in terms of combined static and dynamic memory is identical to the one of DSR with the notable differences that a) TASA doesn't use dynamic memory allocation from heap, and therefore is more certification friendly, b) TASA does not have a bulky runtime system linked with the application and c) TASA does not generate additional code for code or stack randomisation.

3.3.6 Average Performance

Our results regarding average performance, shown in Figure 41, confirm that on average both TASA and Stabilizer provide average execution times close to the default generated code by the compiler when no randomisation is used, with TASA being on average within 0.4% of the default average performance.
In two cases, the average performance of both randomised configurations is significantly different to that of the non-randomised one such as `aifftr` (worse) and `canrdr` (better). This is related to the default memory layout selected by the compiler, which happens to be very good or very bad compared to the set of potential memory layouts, which randomisation can explore. Finally in almost all cases, the execution times of Stabilizer are longer than for TASA, with significant differences at times (`basefp`, `matrix`). The reason is due to Stabilizer's execution time overhead: at start-up the runtime performs some required operations before the execution of the program such as the code relocations, while other additional stack-randomisation are performed at runtime.
Several compiler flags are used to maintain transformations implemented by TASA: (1) `-fno-toplevel-reorder` instructs the compiler to generate functions and globals in the order encountered in the binary; (2) `-fno-section-anchors` prevents placing static variables used in the same function, in nearby locations in order to be accessed with a single base; (3) `-fno-dce` disables dead-code elimination; (4) `-Wnounreachable-code` instructs the compiler to respect the introduced unreachable code padding; and (5) `-Wno-unused-variable` preserves the stack randomisation padding. We have observed that optimisation disabling has neither effect in memory consumption nor in performance. In fact none of the three dead code elimination (dce) passes of the LLVM compiler (dce, advanced dce and global dce) removed any dead or unreachable code. This can be explained because EEMBC have a small/controlled code-base which does not contain any dead code.

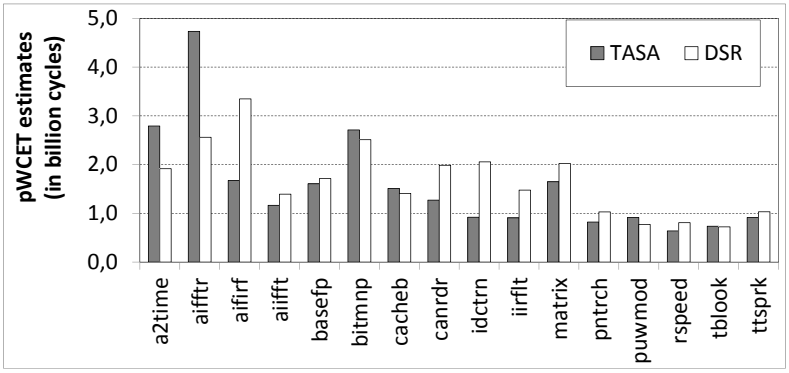

Figure 42: Worst Case Execution Time for TASA and DSR

Table 9: I.I.D. tests for TASA

benchmark	Identical Distribution	Independence
a2time	0.67	0.53
aifftr	0.08	0.16
aifirf	0.33	1.33
aiifft	0.93	0.79
basefp	0.99	0.88
bitmnp	0.83	0.23
cacheb	0.80	1.32
canrdr	0.44	0.88
idctrn	0.65	1.56
iirflt	0.83	0.24
matrix	0.25	0.50
pntcr	0.99	1.13
puwmod	0.99	1.13
rspeed	0.21	0.50
tblook	0.99	1.01
ttsprk	0.66	0.96

3.3.7 pWCET Estimations

In order to compute the pWCET estimations, we use verify that the execution times resulted by using TASA pass the i.i.d. tests as required by MBPTA.
Figure 42 shows the pWCET for EEMBC benchmarks, at a cut-off probability of 10^{-15}, with all possible randomisations enabled w.r.t. the actual execution time on LLVM with no randomisation. For most benchmarks TASA provides lower pWCET than DSR (6% on average across all benchmarks), following the same trend as for average performance, due to the start-up and runtime overhead of dynamic software randomisation. However, in few cases (`a2time`, `aifftr`) the pWCET of TASA is higher than DSR's one despite the lower average performance. This occurs because TASA can explore some rare memory layouts (3% of the explored ones) with much higher execution time than the average one, that DSR was not able to generate, due to its coarser-grain randomisation.

4 Conclusion

In this deliverable we have presented the advances on the research-oriented software solutions for multi-core architectures and their evaluation. First, we have seen the implementation details of the several RTOSes which have been ported to the three hardware target platforms which are supported by PROXIMA. More specifically we have examined the constant-time characteristics of each RTOS variant and how they enable a time-composable basis in the software stack executed on non-MBPTA compliant processors. This provides a two fold purpose: a) it allows the application to be analysed in isolation, without taking into account its complex interactions with the RTOS and b) it facilitates the timing analysis in multi-core processors, such the ones used in PROXIMA.
Second, we presented the software randomisation solutions of each platform and presented a detailed evaluation of their properties. In particular, we have seen the refined implementations of both the dynamic and static software randomisation. We assessed the MBPTA properties added to the software being executed on top of COTS multi-core processors with time-composable operating systems and evaluated in detail the overheads incurred by each randomisation variant. Overall, we have shown that either variant of software randomisation can enable MBPTA on conventional hardware platforms, and that when combined with time-composable RTOSes the analysis of multi-core processors is feasible in a simple, industrial-friendly way.
It is important to note that the amount of the research production of this project is remarkable, given the fact that PROXIMA is an industrialisation-driven project. During the course of this project many research-oriented software solutions have been ported to new platforms showing their potential, while they received significant improvements that will be very useful towards their industrialisation. Moreover, other software components have been developed for the first time, and have reached important milestones such as being evaluated with case studies.
One of the most impressive achievements of the tasks described in this deliverable, is that this work has been successfully completed without the support of an industrial partner, but it was entirely performed by research institutions (BSC, UPD) in a successful way. In particular, the hardware randomisation has received the support of the experienced in the critical real-time domain hardware design firm CG; Timing analysis and tracing support was lead by an expert company in timing analysis, Rapita.
On the other hand the Automotive case study and the successful injection of time-composable properties in open source commercial operating systems have been performed by UPD without being backed by an automotive or RTOS company. Also two software randomisation variants have been made available in several platforms and successfully tested with case studies and commercial operating systems by BSC, without input by a compiler vendor. Therefore, a much higher amount of effort has been invested by those research institutions, in order to be able to reach the same level of maturity level as the hardware randomisation solutions.

Acronyms and Abbreviations

- COTS: Commercial Off-The-Shelf.
- DSR: Dynamic Software Randomisation
- EVT: Extreme Value Theory.
- MBPTA: Measurement-Based Probabilistic Timing Analysis.
- MOET: Maximum Observed Execution Time.
- PRNG: Pseudo-Random Number Generator.
- PTA: Probabilistic Timing Analysis.
- pWCET: Probabilistic Worst-Case Execution Time.
- RTOS: Real-Time Operating System.
- TASA: Toolchain Agnostic Software rAndomisation.
- WCET: Worst-Case Execution Time.

References

[1] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. A time-composable operating system. In *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, pages 69–80, 2012.

[2] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168. ACM Press, 2006.

[3] Alan Burns, Brian Dobbing, and George Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies - Ada-Europe '98, 1998 Ada-Europe International Conference on Reliable Software Technologies, Uppsala, Sweden, June 8-12, 1998, Proceedings*, pages 263–275, 1998.

[4] Alan Burns and Andy J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - mrsp. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 282–291, 2013.

[5] Davide Compagnin and Tullio Vardanega. An automated framework for the timing analysis of applications for an automotive multicore processor. In *21st International Conference on Emerging Technologies and Factory Automation (ETFA), September 6-9, 2016, Berlin, Germany.* (to appear).

[6] C. Curtsinger and E.D Berger. STABILIZER: Statistically Sound Performance Evaluation. In *ASPLOS*, 2013.

[7] http://www.hightec rt.com/en/downloads/sources.html. *Source code of HighTec GPL software.* HighTec.

[8] L. Kosmidis, J. Abella, E. Quinones, and F.J. Cazorla. Efficient cache designs for probabilistically analysable real-time systems. *IEEE Transactions on Computers*, 2014.

[9] L. Kosmidis, J. Abella, E. Quinones, F. Wartel, G. Farrall, and F.J. Cazorla. Containing timing-related certification cost in automotive systems deploying complex hardware. In *DAC*, 2014.

[10] L. Kosmidis, C. Curtsinger, E. Quinones, J. Abella, E. Berger, and F.J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.

[11] L. Kosmidis, R. Vargas, D. Morales, E. Quinones, J. Abella, and F.J. Cazorla. Tasa: Toolchain agnostic software randomisation for critical real-time systems. In *ICCAD*, 2016.

[12] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th ASPLOS*, pages 265–276, 2009.

[13] E. Quiñones, E.D. Berger, G. Bernat, and F. Cazorla. Using randomized caches in probabilistic real-time systems. In *Proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[14] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Special publication 800-22rev1a, US National Institute of Standards and Technology (NIST), 2010.

[15] T. van Aardenne-Ehrenfest and N.G. de Bruijn. *Circuits and Trees in Oriented Linear Graphs*, page 149.

[16] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F.J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.

[17] Marco Ziccardi, Alessandro Cornaglia, Enrico Mezzetti, and Tullio Vardanega. Software-enforced interconnect arbitration for COTS multicores. In *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, pages 11–20, 2015.