A Study in Specification and Hardware Runtime Verification of Critical Embedded Software

Dimitry Solet, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Sébastien Pillement

A Study in Specification and Hardware Runtime Verification of Critical Embedded Software

Dimitry Solet, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Sébastien Pillement

Abstract—We evaluate HARVEST (Hardware Accelerated Runtime Verification for Embedded SofTware), a runtime error detection mechanism for embedded software running on standard SoPC architectures, *i.e.*, one (or more) processor(s) and an FPGA on a single chip. The program under monitoring runs on the processor(s), while a trace analysis module running on the FPGA detects errors in its execution. The hardware implementation of trace analysis minimizes the performance impact and achieves a very low error reporting latency (a few processor cycles). In this article, we evaluate the suitability of using HARVEST to detect errors resulting from transient physical faults.

We explain how HARVEST is used to monitor a complex software component (Trampoline RTOS) on a commercially available hardware platform (Microchip SmartFusion2). We measure the overhead of the resulting instrumentation on system resources. We then evaluate its performance in detecting silent data corruptions, based on a systematic simulation of bit flips at the instruction set architecture level. We report a detection rate of up to 90.2 % for a fairly low system resource overhead, suggesting an interesting trade-off for designers of highly constrained critical systems.

1 Introduction

With the evolution of technologies, and in particular increased transistor density, the sensitivity of integrated circuits to physical phenomena such as cosmic rays increase. Increasing the susceptibility of embedded systems to single-event effect (SEE) leads to an increase of faults such as bit flip at runtime. This motivates the use of runtime detection and mitigation techniques.

Existing error detection techniques rely on some form of redundancy at the hardware and/or at the software levels. They try to find a trade-off between efficiency (ability to detect errors), accuracy (false positive rate), impact on design cost, impact on production cost, and impact on system performance. In this context, System-on-a-Programmable-Chip (SoPC), *i.e.*, architectures that integrate on the same chip one (or more) processor(s) and FPGA fabric, offer interesting opportunities in hardware accelerated runtime monitoring.

In this work we propose a runtime error detection mechanism that exploits the possibilities offered by such platforms. The proposed mechanism detects errors that appear during the execution of the embedded software running on the processor(s) of the SoPC. To minimize the impact on performance, the trace analysis is performed on the FPGA of the SoPC. To do so, dedicated monitors are automatically generated from a formal specification capturing the expected properties of the embedded software. Thanks to the hardware acceleration of the analysis, the achieved error signalling latency amounts to a few processor cycles.

- *D. Solet is with ESEO - France. E-Mail: dimitry.solet@eseo.fr*
- *S. Pillement is with Nantes Université - UMR 6164 IETR, F-44300 Nantes France. E-Mail: sebastien.pillement@univ-nantes.fr*
- *JL. Béchennec, M. Briday, and S. Faucou are with Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000 Nantes, France. E-Mail: firstname.name@ls2n.fr*

For this study, the software under verification is Trampoline [1] Real Time Operating System (RTOS), an open source implementation of the AUTOSAR Classic Platform Operating System standard. It is a relatively complex software component that is typically found in real-time embedded control systems. The kernel is the most critical part of the system, it runs in supervisor mode, with access to all RAM - an error can affect all application tasks.

We specified its behavior using a set of properties expressed in temporal logic. These properties are used to generate the monitors needed for the error detection mechanism. We slightly instrumented the code of Trampoline and adapted the link scripts to make it easier for the detection mechanism to observe the execution. We implemented the system on the SmartFusion2 off-the-shelf SoPC[1]. We then exposed the resulting system to a systematic fault injection campaign simulating single event upsets that result in bit inversions at ISA register level[2]. The results obtained indicate that the detection mechanism offers an interesting balance between efficiency (between 75.2 % and up to 90.2 % of silent data corruptions are detected) and performance impact (in a configuration offering a detection rate of 84.8 %, the maximum impact measured on RTOS latency is 38.1 %). The result also show that the FPGA resources required by the RTOS application fit within the scale provided by commercial SoPCs. This allows the remaining hardware resources to be used either for runtime verification of the application part or for a more traditional use, *e.g.* accelerators.

: In Section 2, we provide context and present related work. In Section 3, we describe the design and architecture of HARVEST. In Section 4, we explain how we used HARVEST

1. Microchip Smartfusion2: https://www.microchip.com/en-us/products/fpgas-and-plds/system-on-chip-fpgas/smartfusion-2-fpgas

2. NVM and SRAM of the SmartFusion2 already support Single bit Error Correction and Dual bit Error Detection (SECDED), this is why fault injection is limited to ISA registers.

to monitor Trampoline RTOS. In Section 5, we describe the experimental protocol. In Section 6, we summarize the results of the fault injection campaign and discuss our main findings. In Section 7, we draw conclusions from this work and propose some possible extensions.

2 Context and related works

2.1 Runtime Monitoring

Runtime monitoring aims to check online the correct behavior of a system. A runtime monitoring framework is usually composed of four stages:

- An *observation stage* captures in real-time a set of execution events.
- A *state identification stage* processes the events to compute the evolution of the state of the system.
- A *verification stage* checks that the trace formed by the sequence of states conforms to the specification.
- A *notification stage* raises a signal when a violation of the specification is detected.

The observation stage is always interfaced with the system during its execution. For the other stages it depends on the purpose of the monitoring. If the purpose is to debug the system or to study its performance these stages can appear offline (post-mortem). But if the purpose is to detect runtime errors, then all stages must be interfaced with the system at runtime. In this work we focus on this latter case.

A runtime monitoring framework can be implemented as software, hardware, or a combination of both. In the software implementation the code of the framework is typically weaved with the code of the program under monitoring. This offers a straightforward observation of program-level entities (*e.g.*, threads, or variables) and allows to express complex properties easily. Unfortunately, the resource consumption of such implementation scales linearly with the number of properties, both in memory and time like with Enforcer [2] or Copilot [3]. This can be a serious problem for systems with a limited amount of resources and especially for real-time embedded systems that have stringent memory and timing constraints. A hardware implementation of the framework capable to checking multiple monitors in parallel is a possible solution to this problem.

Hardware implementation can be used to monitor both hardware components [4], [5] or programs [6], [7]. In this work we focus on the monitoring of software component running on top of a possibly faulty hardware. In this case, it is usually assumed that the program runs on a soft core processor [6], [7] with probes added to the core design to observe low level signals (*i.e.*, memory transactions). The main difficulty is then to rebuild the system state from these signals. One possibility is to use debugging information as explored in [7] to relate the memory locations to the program symbols. This is important to allow the expression of properties at the appropriate level of abstraction, *i.e.*, most of the time at the source code level.

Other authors have focused on hardware implementations that do not require modification of the computation core [8], [9]. The target architecture in that case is a SoPC that integrates a processing core and an FPGA fabric. The processing core hosts the program under verification and the FPGA hosts the monitoring framework. The question is now how to observe the execution of the program and how to transfer this observation to the monitoring framework. In the literature two directions have been explored.

One is to use the built-in debugging/tracking module such as the ARM CoreSight technology that allows to send low-level execution events to a dedicated memory or over a bus. This is the case of ARMHeX [8], a dynamic information flow tracking (DIFT) framework. The use of ARMHeX requires to slightly instrument the source code of the program under monitoring and to set restrictions on the compiler back-end to avoid constructs that impair observability. As in the case of instrumented softcore, the use of low-level events makes it more difficult to express properties at the correct abstraction level. Moreover, built-in debugging/tracking modules can usually observe many execution events but not all of them. Monitoring frameworks built using this approach need to overcome these limitations.

The second direction is to integrate the monitoring framework as a classic peripheral device [9]. In this case the program has to explicitly send data or events over a bus to request a verification. This second approach is generally more intrusive and has a higher overhead on the execution time but it is also more flexible as it is not restricted to observe low-level execution events and is less dependent to a specific technology. This is the approach used in HARVEST.

2.2 Runtime Verification

Runtime verification (RV) is the branch of formal methods dealing with the synthesis of monitors from formal specifications [10]. Looking at the architecture of a monitoring system, it quickly becomes apparent that monitors are the most functionally complex part. The ability to use components that have been synthesized using a tool chain based on formal methods is therefore a way to significantly improve the reliability of the system.

RV techiques use various formalisms to specify the properties of the system under verification: (extended) regular expressions [11], temporal logics (LTL [12], ptLTL [13]), real-time logics (TLTL [12], Mission Time-LTL [14]), and specification languages such as PSL [15] which brings together several of the previous formalisms. For each of these formalisms, one or more synthesis algorithms have been proposed that automatically construct a monitor for the specifications. This monitor usually takes the form of a state machine or a Boolean circuit. For HARVEST, we adopt ptLTL as the specification language and Boolean circuits as the target of synthesis.

2.3 Robust RTOS

In a real-time system, applications run on top of an RTOS, which arbitrates the allocation of system resources to tasks so that all deadlines are met. Failure of the RTOS can result in a deadline violation, which in turn can lead to a system-level failure. The central role of the RTOS justifies the interest in monitoring and enforcing its runtime behavior.

In [16] Rodriguez *et al.* introduce the idea of robust RTOS. In their work, the key concept is to wrap the RTOS kernel with a runtime verification layer that checks properties whenever a system call returns. These wrappers are automatically generated from temporal logic formulas that specify the expected behavior of the kernel as observed from the application level. Since the wrappers are in user space, errors are only detected

when they propagate through the kernel/application interface, which may already be too late to trigger a fine-grained recovery mechanism. In general, detecting and isolating errors as early as possible facilitates the recovery process. We pick up the idea of automatically generating monitors from a formal specification of RTOS behavior. The use of an open source RTOS allows us to integrate the runtime verification mechanism in the kernel. This enables us to express properties at a finer grain, and to minimize the error signaling latency.

In [17] Hoffman *et al.* investigate the design of a fail-safe RTOS called dOSEK, a dependability-oriented static RTOS. Static analysis of the application is used to specialize the code of the kernel in order to minimize the amount of memory and registers exposed to transient hardware faults. Then, specific coding practices are used together with a fine-grained arithmetic encoding of kernel data structures to minimize the rate of residual errors. The results show a sharp reduction of silent data corruptions down to four orders of magnitude, at the cost of a significant overhead on key performance indicators such as the kernel latency (up to ×6.3). Our approach has less impact on the RTOS code. Moreover, we offload the most expensive part of the computation to a hardware accelerator. As a result, the impact on the kernel latency is much lower. On the other hand, the detection rate is lower. The Hoffman *et al.* and HARVEST approaches are incomparable, each making very different tradeoffs between detection rate and overhead.

3 HARVEST

3.1 Overview

HARVEST [9] is a hardware-accelerated runtime monitoring framework for embedded software. It has been initially designed to overcome the scalability issues of a software monitoring framework [2] that was designed to verify high-level properties, for instance complex communication patterns between threads. This software framework has to verify the monitors sequentially, which results in a linear increase of the traversal of the verification stage as a function of the number of monitors. The promise of the hardware approach is that all monitors can be checked in parallel, keeping the verification step traversal time low and error-reporting latency independent of the number of monitors.

Since offloading verification to an accelerator solves scalability problems "by design", the question we investigate here is whether this also allows us to handle finer-grained properties expressed at the source code level, or even at the machine code level. Working at this level would allow, for example, extending the fault model covered by the framework to hardware faults. One of the goals of this work is therefore to precisely define the type of property for which HARVEST provides a relevant monitoring mechanism.

3.2 Specification of the properties to be monitored

The first step in using HARVEST is to identify and specify the expected behavior of the program under monitoring as a set of ptLTL formulas. It is possible to monitor both coarse-grained and fine-grained properties. An example of a coarse-grained property is: "*Any data produced by thread A is consumed by thread B*". An example of a fine-grained property is: "*In function f, variable X must always have a value in the interval [0, 10]*".

3.3 Architecture

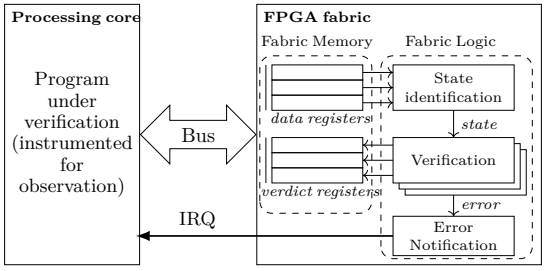

Fig. 1. Architecture of HARVEST. Monitors are located in the FPGA and consist of 3 steps in a row: state identification, verification and error notification.

Figure 1 is an overview of the architecture of HARVEST. The program under monitoring and the observation stage run on the processor unit. The following stages of the monitoring framework run on the FPGA fabric. The memory area of the FPGA acts as an interface between the observation stage and the following stages. Arrows between these stages indicate the data flow.

The following paragraphs provide details on each stage, using the following (fined-grained) property as a running example:

variable `foo` *must be set to 1 during the function* `func`

3.4 Observation stage

Once the properties are specified, it becomes possible to deduce the information that the monitoring framework needs to observe: this is the information needed to evaluate the atomic propositions used in the ptLTL formulas corresponding to the properties. The program under monitoring must be instrumented to extract this information and send it to the following stages. As illustrated in figure 1, this is done by writing data to memory locations in the FPGA. Two complementary approaches are used to extract the information, one for control flow monitoring and the other for data flow monitoring.

A control flow monitoring event (*e.g.*, function call or function return) requires the insertion of a write instruction into the program. This instruction sets a certain bit of a certain data register. This bit is statically assigned to this particular control flow event. This same bit is reset by hardware when the event is acknowledged.

For data flow monitoring, we currently rely on mapping the variable of interest directly to the FPGA memory. This solution has the advantage that no additional instructions are introduced into the program, only the variable declaration and the link script have to be adapted. However, it is reserved for a subset of variables because the number of data registers is limited. Also, to keep track of all updates, every write must be done in memory. This behavior is obtained by using the C language keyword *volatile*.

The overhead of instrumentation depends on the number of observation points. For control flow, the insertion of new instructions increases both code size and execution time. For data flow, only the execution time is increased. The increase in execution time is mainly due to the fact that memory-mapped IOs are usually slightly slower than regular memory accesses, especially on systems with a data cache. Also, the use of the

volatile keyword prohibits compile-time optimizations that promote a variable to a CPU register. Both the timing and memory overheads of full instrumentation of the Trampoline RTOS are evaluated in Section 4.4.

In our running example, we need to observe variable `foo`, and the control flow events linked to function `func`. At the time of declaration, variable `foo` variable must be mapped in a memory section of the FPGA. Assuming that the `.fabricMem` is declared in the link script, the declaration with *GCC* C compiler syntax would be:

```
volatile int foo
  __attribute__ ((section(".fabricMem"))) = 0;
```

To observe the entry of control flow into the function `func`, we simply place a write operation at the beginning of the function. Conversely, to observe the exit of control flow from the function, we place a write operation at the end. The targets of these writes are different bits of a volatile variable, also mapped in the `.fabricMem` section.

3.5 State identification stage

State evaluation consists of processing the information available in the data registers to extract a symbolic state that can be more easily used in the next stages. Most of the time it consists of a combinatorial expression that outputs a Boolean information.

For now, this step must be written by hand using a hardware description language (VHDL in our case). Nevertheless, HARVEST provides a library of circuits corresponding to the basic comparators and arithmetic operations. In practice, on a real system, this step remains very simple.

For our running example, three Booleans are sent to the next stage: values of bits $func_{in}$ and $func_{out}$, and the output of the comparison $foo = 1$, where $func_{in}$, $func_{out}$ and foo are the data written in the observation stage.

3.6 Verification stage

The *verification* stage implements the monitors which are automatically synthesized from the ptLTL formulas. Currently, HARVEST implements the synthesis algorithm of Havelund and Rosu [13]. Each monitor evaluates one ptLTL property, and all monitors are evaluated in parallel. Each monitor takes as input a set of Boolean values associated to the atomic proposition present in the formula. All these Boolean values form the symbolic state of the system as computed by the previous stage.

An elementary ptLTL operator is a sequential circuit. Beside the Boolean values associated to atomic propositions, this circuit also takes as input its output of the previous cycle. It is typically composed of a few logic gates and a D flip-flop to store the output. These elementary operators are then connected according to the syntax tree of the formula. Each output of a formula is associated with a bit in one of the verdict registers of the FPGA. If a formula is not valid the verdict register is no longer zero and an error signal is sent to the notification circuit (see *error* in Figure 1). The VHDL code for this stage is entirely generated from the system specification, thanks to a Python script.

When the latency of the critical path through the identification and verification stages is greater than the period of the system clock, HARVEST allows to pipeline the execution of these stages. This feature has not been required so far, including in the monitoring of a real system presented in the next sections.

In our running example, the property is expressed by formula 1.

$$\boxdot(\uparrow (foo = 1) \rightarrow [func_{in}, func_{out})_s) \quad (1)$$

The $\boxdot$ character means that the property is always true, *i.e.* throughout the execution trace. The $\uparrow$ character indicates that the Boolean expression is true, but was false in the previous cycle. In other words, the expression *becomes* true. The expression $[a, b)_s$ becomes true when we observe an occurrence of `a` and remains true until we have an occurrence of `b`. So, the corresponding ptLTL formula is: Whenever the variable `foo` takes the value 1, we must have seen an occurrence of $func_{in}$ but no corresponding occurrence of $func_{out}$.

3.7 Error notification stage

The error notifier is the simplest component. It simply generates an interrupt when the error signal becomes true (*i.e.*, the property becomes false). This notifier depends on the underlying architecture, this is why it is separated from the verification stage.

When the CPU handles the interrupt (at the software level), it can read the verdict registers (memory area in the FPGA) to determine which formula(s) failed and trigger an appropriate recovery action.

3.8 Implementation on SmartFusion2 SoPC

The HARVEST architecture is implemented on top of the SmartFusion2 SoPC that integrates an ARM Cortex M3 micro-controller and an FPGA with 60K logic elements (each 4-LUT with DFF). An AMBA bus interconnects the microcontroller and the FPGA. This bus supports the AHB mode for high performance operations (bursts, pipelines) and the APB bus for common operations (reading and writing registers). The implementation uses APB as it is sufficient regarding the requirements.

In the following experiments, the detection latency is limited to 5 clock cycles: 2 cycles are required for the write transaction on the APB bus, 1 cycle for the traversal of the state identification stage, 1 cycle for the traversal of the verification stage, and 1 cycle for interrupt generation. In practice, the microcontroller is only stalled when writing to the registers. Thus, by default, the IRQ signaling an error is generated 3 cycles after the commit of the instruction used to send the data to the FPGA. For some systems, it may be necessary to ensure that no instruction is executed before the verification verdict is reported. A first simple software-only solution is to add NOP instructions after each write instruction to the FPGA data registers. A second solution uses the APB protocol acknowledgement signal to block the microcontroller until the end of the verification cycle. This is a more efficient way to implement the solution, but it is hardware dependent.

4 Runtime verification of Trampoline RTOS

We used Trampoline RTOS, a free and open source[3] RTOS compliant with the AUTOSAR OS 4.2 standard [18]. Trampoline RTOS is dedicated to deeply embedded systems in the automotive industry where dependability is a key factor. This type of RTOS is called static because the kernel does not provide services to dynamically create system objects: each object must be declared at compile time. This approach allows to tailor the kernel according to the static requirements of the system by eliminating useless services and enabling static allocation of kernel data structures. A static approach inherently increases dependability [17] with the reduction of vulnerable run-time states which limits the exposition to SEU.

4.1 Basic behavior of a system service

The scheduling part of the Trampoline kernel uses essentially three data structures. The first is the set of task descriptors. The second is the list of ready tasks. The third is named *tpl_kern*. It gathers information about the running task and the elected task, *i.e.* the task chosen by the scheduler to replace the running task. This information is composed of the identifier of the task, a pointer to its static descriptor and a pointer to its dynamic descriptor. *tpl_kern* also contains 3 flags: *need_schedule* (a rescheduling is required), *need_switch* (a context switch is required) and *need_save* (the context of the preempted task shall be saved).

In Trampoline, the execution of a service is basically divided into four main phases. The first phase is service specific. It updates the kernel data structures (adding a task in the ready list, changing the state of the running task, ...). The second phase depends on the value of the *need_schedule* flag. It calls the scheduler which selects the highest priority task and modifies the associated data structures (*elected task*, *need_save* and *need_switch* flags). The third phase depends on the value of the *need_save* flag. It saves the context of the current task. The fourth and final phase depends on the value of the *need_switch* flag. It performs the context switch.

4.2 Specification of the expected behavior

The first step in the monitoring of the kernel of a RTOS is to capture its expected behavior as a set of ptLTL formulas. Of course, it is neither possible nor desirable to capture the whole behavior of the kernel. The main goal is to increase the robustness of the kernel against SEU and, in particular, to detect errors that could lead to incorrect scheduling decisions. The kernel is monitored at two complementary levels: 1) The state of *tpl_kern*; 2) The phases of the execution of a service; 3) The control flow inside the kernel.

4.2.1 Monitoring the state of tpl_kern

Let us define the state S of the *tpl_kern* data structure: $S_{\{need\_schedule,need\_save,need\_switch\}}$ (concatenation of the three flags). Thus, S_{001} is the state in which only *need_switch* is true. The analysis of the source code of the RTOS allows to model the evolution of the states of the kernel as illustrated in Fig. 2.

3. https://github.com/TrampolineRTOS/trampoline, instrumented version for Harvest is available in branch `cortex-instrumented`.

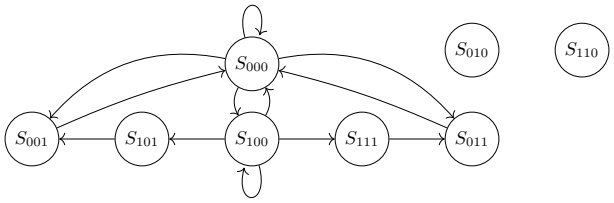

Fig. 2. Possible evolution of the flags of the kernel states. The state identification is the concatenation of the 3 flags *need_schedule*, *need_save* and *need_switch*.

Some properties can be extracted from the evolution of the kernel state:

Safety property: Checking that the kernel is not in an unreachable state. As shown in Fig. 2, states S_{010} and S_{110} are unreachable. This is explained by the fact that in the absence of a context switch, the context does not need to be saved.

Reachability property: Check the consistency of the state sequence taken by the kernel. This means that when the kernel enters a state it was previously in an allowed predecessor state. For instance, if the current state is S_{001}, the previous state was either S_{000} or S_{101}.

Invariant property: When the kernel enters in state S_{100}, it means that the scheduler should be running. The pointers to both the running process and the elected process have not yet been updated (they are identical) and should point to the same object.

4.2.2 Monitoring the execution phases of a service

When the running task emits a system call, the system switches to kernel mode and executes the System Call Handler (SCH). As explained in section 4.1, the SCH has to (1) call the required service, (2) call the scheduler if needed, (3) save the context of the running process if needed, (4) perform the context switch if needed.

To monitor the execution of the SCH execution, we define the following atomic propositions:

- run_elec: pointers to the running task and the elected task have the same value.
- $call\_handler$: SCH is being executed.
- $service\_OS$: service is running.
- $context\_switch$: context switch is required.
- $save\_context$: context save is required.

The requirements to be checked are:

- run_elec is always true except during the execution of the SCH. Its value is therefore tested at the beginning and at the exit of the SCH:
  $\boxdot(\uparrow call\_handler \rightarrow run\_elec)$
  $\boxdot(\downarrow call\_handler \rightarrow run\_elec)$
- At the start of the service call, the kernel must be in S_{000}. At the end, *need_schedule* shall be equal to 0. Therefore the kernel cannot be in S_{101} or S_{111}:
  $\boxdot(\uparrow service\_OS \rightarrow S_{000})$
  $\boxdot(\downarrow service\_OS \rightarrow (\neg(S_{101} \vee S_{111})))$
- Context switch is performed if *need_switch* is set. Therefore the kernel should be either in S_{001} or S_{011}. At the end of the context switch, run_elec must be true:

$\boxdot(\uparrow context\_switch \rightarrow (S_{001} \vee S_{011})$
$\boxdot(\downarrow context\_switch \rightarrow run\_elec)$

- The context of the previous running task is saved if *need_save* is set and *context_switch* is true: $\boxdot(\uparrow save\_context \rightarrow (S_{111} \wedge context\_switch)$

An atomic proposition is associated with each service and is set when it is executed. It allows to define service-specific properties. For example, the *ActivateTask* service allows to activate a new instance of a task (add a job to the *ready list*). When it is run, only 4 states are reachable: S_{000}, S_{100}, S_{111}, and S_{011}. The initial state is S_{000} as set by the SCH. If the activation is illegal, the service stops and terminates at S_{000}. Otherwise *need_schedule* is set (S_{100}) and a schedule is executed. If the running task still has the highest priority, no context switch is required and the service will stop at S_{100}. Otherwise, the context of the running task should be saved, state S_{111}. Since the *need_schedule* bit is reset after the *elected process* is updated, the final state is S_{011}. This analysis is then translated into two ptLTL properties.

The same analysis is performed for all services that can modify the state of the kernel.

4.2.3 Monitoring of kernel control flow

A set of properties is used to check that the control flow within the kernel is valid with respect to the call graph. Each internal function is associated with an atomic proposition, which is set at function entry and reset at function exit. For each function that is not a top-level function in the call graph, a property states that whenever it is running, then one of its parents in the call graph is also running. This approach assumes that there are no recursive calls, which is a relatively safe assumption in the context of critical embedded systems.

4.3 Instrumentation of kernel code

The principles described in Section 3.4 form the basis of the instrumentation of Trampoline RTOS. Kernel data structures are mapped to data registers into the FPGA. Since FPGA mapped memories are only accessible in 32-bit words, the relevant kernel data structures have been redesigned so that all elements are 32 bits wide. These mappings allow to track the state of the kernel and the associated atomic propositions. For the atomic propositions corresponding to control flow events, instructions are added to set the bits assigned to each event.

4.4 Overhead evaluation

The instrumentation of the kernel increases the execution time of system calls (temporal overhead) and increases the memory footprint of the code (memory overhead).

4.4.1 Temporal overhead

The measurement of the temporal overhead is obtained from the execution time of each service with and without instrumentation. The execution time of each service is measured on a SmartFusion2 running at 142MHz with a code optimization level `-O0` and a standard configuration of Trampoline.

Results presented in Table 1 highlights an overhead between 15.2 % and 38.1 % depending on the service. Considering that the execution time of a system call is usually negligible compared to the application execution time, the measured overhead can be considered negligible too. This limited overhead is required to observe the software execution and is not related to the evaluation of the monitor: *i.e.* it does not depend on the number of monitors, but on the number of observation points. In comparison, a pure software solution applied to a comparable RTOS [17] results in a time overhead between 60 % and 630 %.

OS service	Execution time		Overhead
	vanilla	instrumented	
ActivateTask †	4.9 μs	6.5 μs	32.6%
ActivateTask	17.2 μs	22.6 μs	31.4%
TerminateTask	5.4 μs	7.4 μs	37.0%
ChainTask	12.8 μs	16.7 μs	30.4%
Schedule †	3.4 μs	4.6 μs	35.3%
Schedule	13.5 μs	17.7 μs	31.1%
SetEvent †	3.3 μs	3.8 μs	15.2%
SetEvent	11.8 μs	15.6 μs	32.2%
WaitEvent †	2.1 μs	2.5 μs	19.0%
WaitEvent	6.9 μs	9.3 μs	34.8%
GetResource	2.1 μs	2.9 μs	38.1%
ReleaseResource †	1.9 μs	2.6 μs	36.8%
ReleaseResource	8.4 μs	11.5 μs	36.9%

TABLE 1
Temporal overhead for each service. Some services may trigger a context switch. These services are thus presented twice: once without context switch (suffixed with †) and once with context switch.

4.4.2 Memory overhead

To measure the impact of the instrumentation on memory footprint, we added the size of the section containing kernel code with constants for an application that calls all of the services listed in Table 1. It increases from 6436 bytes to 7079 bytes with instrumentation (less than 10 % increase). This is quite a small overhead even for a small RTOS.

5 Evaluation framework

This section introduces the evaluation framework, it includes a description of the fault injection environment and the requirements for the application under test.

5.1 Fault injection environment

Fault injection techniques for integrated circuits are generally classified into 3 categories: hardware-based, emulation-based, and simulation-based fault injection approaches [19]. The hardware-based approach faithfully reproduces the physical phenomena that cause faults. However, this technique is difficult to use and provides poor controllability and observability. Emulation and simulation-based techniques require a low-level model of the hardware architecture, usually FPGA-based for the former and software-based for the latter. They allow injections anywhere in the circuit. These two approaches are not appropriate in our case because we don't have a detailed model of the hardware platform.

For programmable embedded systems, in contrast to many simulation approaches based on QEMU [20] or GEM5 [21], a hybrid approach based directly on the real hardware is feasible. In this case, the error is injected through the software, either by instrumenting the code [22], [23] or by using a debugging probe [24]. The main drawback of such an approach is that fault injection is limited by design to resources accessible

by software, *i.e.* it is not possible for instance to inject a fault into a CPU pipeline register.

This approach is temporally intrusive, but is fairly easy to implement and allows reproducibility of the fault injection campaign. As the test application does not interact with the environment, temporal intrusiveness is not a problem and this approach seems to be the best in our case.

5.1.1 Classification of the results

The results of the fault injection campaign are classically defined in 4 categories [25], [26]:

No impact. The fault does not affect the program execution and the program terminates normally and the output is correct.

Exception. The fault produces an error detected by the hardware mechanisms which raises an exception.

Timeout. The fault produces an error that prevents the program from completing. A watchdog is required to detect this type of error.

Silent Data Corruption (SDC). The program terminates but the output is incorrect. The consequence of a SDC is a deviation from the expected behavior of the system.

The detection of the SDC is the main objective as a category 1 fault has no impact and category 2 and 3 are already detected by hardware mechanisms.

5.1.2 Fault model

When a particle beam hits the memory or a register, several contiguous bits may be affected by a bit flip. However, the authors of [26] show that the results obtained with the single bit flip model and the double bit flip model are close. In [23], a model that allows multiple bit inversions shows that, in a limited number of cases (8%), the number of SDC is increased. Therefore, we assume that the use of the single-bit flip model is still relevant and limits the explosion of the fault space.

In the single bit-flip fault injection model, the fault space is defined by a set of time-location tuples where time is the date of the fault injection and location is a specific bit in a register or a memory word.

We use the *inject-on-read* [25] technique to reduce the number of fault injections. Intuitively, a fault injection on a memory location is performed only during a read access. This injection produces the same result as any fault injection that occurred in the same memory location between the previous access (read or write) and the current read. At the analysis level, the access is weighted according to the exposure time, *i.e.* the time the data is stored between the 2 accesses.

In our case, the platform considered (SmartFusion 2) targets safety-critical systems and both embedded SRAM and non-volatile memory support a single-bit error correction and dual-bit error detection code (SECDED). As a result, the faults considered are limited to the CPU registers and the fault space is significantly reduced.

Note that SmartFusion 2 embeds an SEU immune Zero FIT (Failure In Time) Flash FPGA configuration, and the D-flip-flops used by the monitors are protected by Triple Modular Redundancy (TMR). As a result, runtime monitor failures are not taken into account.

5.1.3 Fault Injection Implementation

The fault injection platform is based on a host machine that controls the target (*i.e.* the SoPC) through a GDB debugger interface. A custom script controls the debug probe (J-Link) to interact with the SmartFusion2 board using a JTAG link.

The fault injection campaign is performed in two steps. First, the application is executed instruction by instruction (at the assembly level) to determine the reference trace (named golden trace or *gt*). This implementation approximates the time spent by the number of assembler instructions executed along the trace, noted L_{gt}. This trace is analyzed to determine the read/write accesses for each register (required to implement the *inject-on-read* technique).

The second step is fault injection. If we consider an exhaustive fault injection campaign on 1 32-bit register, then there are $L_{gt} \times 1 \times 32$ time-location tuples to evaluate. For each of these cases, simply 1) execute the application up to the moment of injection (insert a breakpoint along the execution trace corresponding to the time of the tuple) 2) perform a fault injection, using the debug probe, to flip a bit of the target register, corresponding to the location of the tuple 3) resume the application. The execution trace of this run is collected and compared to the golden trace *gt*.

5.2 Application

The constraints of functional fault injection with a debug probe require that the application be reproducible (not dependent on external phenomena) and not time sensitive (injection takes a long time and can interfere with the timing of the application). In addition, the analysis is performed by comparison with a reference trace and this trace must be finite; the application must therefore have a beginning and an end.

The main goal is to make the kernel services more robust, especially the scheduler. This leads to designing an application that makes most of the kernel system calls that are not related to interrupts. It is unrealistic to evaluate every possible control flow for every service. However, we distinguish between services that can perform rescheduling. For example, the `ActivateTask` service inserts a task into the ready list. If the inserted task has a higher priority than the current task and the current task is preemptible, then a rescheduling is performed. In the other case, the current task resumes its execution. For each system call that is likely to cause a context switch, the application implements the 2 options. The set of service calls is summarized in Table 1. The application requires 6 tasks, each of which performs 1 job[4]. There are 7 preemptions in total.

After each system call, a checkpoint verifies that the scheduler's behavior conforms to the expected result, thus preventing 2 consecutive errors from leading to an apparently correct result. There are 21 checkpoints, and a scheduling error changes the order in which these checkpoints are called. In this way, "*no impact*" and *SDC* results can be distinguished by comparing the order in which the application passes through the various checkpoints to the reference trace.

The application is run 3 times in a row to ensure that an injection that disrupts the scheduling later is covered

4. The full code of the application is available at https://github.com/TrampolineRTOS/trampoline/tree/cortex-instrumented/examples/cortex/armv7m/SmartFusion2/starterKit/FI_appli

(error propagation latency). We have limited our experiments to 3 runs because in our experiments, 92.4% of the SDC occurs during the first run, 7.6% during the second run, and none during the third run. The checkpoint order is stored in an array that is tripled. This prevents fault injection from corrupting this array, even if the scheduling is correct.

A first fault injection campaign has been published in [27]. The monitors were constructed from 49 ptLTL formulas for the runtime verification of the RTOS scheduler behavior. This first evaluation resulted in an SDC detection rate of 48.4%. In this work, we extend the set of formulas to cover a broader range of the RTOS behavior in order to increase the detection rate.

6 Evaluation of the Runtime Verification strategy using Fault Injection

This section provides a detailed analysis of the results of the fault injection and the results of the verification mechanism. Based on these results, an iterative process is introduced to improve the detection mechanism in order to take into account a larger set of cases. This analysis evaluates the potential and limitations of a hardware verification mechanism associated with an off-the-shelf SOPC-based architecture.

6.1 Overview of the evaluation process

6.1.1 Configurations

Fault injection is performed on 4 different configurations of the application to find an appropriate trade-off between the detection rate of the SDCs and the instrumentation code overhead required to achieve the monitoring. These 4 configurations are detailed in the following sections.

For each configuration the golden trace L_{gt} is related to the execution of the reference application defined in Section 5.2. As this trace includes the instrumentation code, it might differ from one configuration to another, as explained below.

All the configurations are compiled with gcc (`arm-none-eabi-gcc`) version 4.9.3 without any optimization (`-O0`).

6.1.2 ARM registers analysis

The Cortex M3 architecture uses the ARMv7 Thumb2 instruction set architecture based on 16 32-bits registers `r0` to `r15`. Registers `r13` to `r15` are reserved for the stack pointer (`sp`), the link register (`lr`) and the program counter (`pc`) respectively. The link register is used to store the return address during a function call. For nested calls, it is necessary to save the link register on the stack. The `r7` register is the stack frame pointer and is used locally as a stack pointer within a function.

The stack is used when a function is called. It hosts the frame of the function where local variables are allocated and callee-saved registers are saved. Function parameters are passed in registers `r0-r3` and possibly on the stack if their number or size is too large. In our case, the internal kernel functions take a maximum of 3 parameters and use only the registers.

The general registers `r0` through `r5` are used for both function parameter storage and general program execution. The registers `r6` and `r8-r12` are not used by the compiler in the kernel code. They are only saved/restored during context switches. All injections into these registers will have no impact on the scheduling.

Considering a time resolution at the assembly instruction level (L_{gt} instructions), sixteen 32-bit registers, the number of 1-bit flip fault injections required for an exhaustive study increases to $16 \times 32 \times L_{gt}$.

6.2 Results of the fault injection campaign

6.2.1 Injection without activating the monitoring

First, a fault injection campaign is performed without activating the monitoring but with the instrumentation code to work on the same golden trace. The results of this first reference injection are detailed in Table 2.

		details for some registers			
	Global	**r7**	**sp**	**lr**	**pc**
No impact	79.7%	23.5%	67.6%	14.3%	12.3%
Exception	18.7%	75.3%	31.6%	80.3%	83.4%
Time out	0.2%	0.6%	0.4%	1.9%	0.5%
SDC	**1.4%**	0.6%	0.5%	3.4%	3.9%
	99,573	2,514	2,062	14,940	17,172

TABLE 2
Distribution of the faults for each category. The number of faults is given in addition to percentages for Silent Data Corruption category.

These results show that the majority of fault injections have no impact or are detected by a hardware mechanism that raised an exception. Only 1.4 % of faults lead to an SDC. In our case this means that 1.4 % of the injections in the kernel corrupt the behavior of the scheduler without any exceptions or timeouts.

These general results can be refined by looking at the register types. Injections on `pc` or `lr` generally lead to an exception. For `pc`, there is always a hardware exception when injecting faults on bit 14 and above. Injections on bit 0 have no impact (instruction codes are on 16 or 32 bits, and the `pc` value is expected to be even). This is the same for `lr` as it stores the return address of the calling function. Injections on `sp` (stack pointer) lead to a large number of non-impact faults. Since compiler optimizations are disabled, addressing of local variables is based on the frame pointer instead of the stack pointer. The register `r7` is used to host the frame pointer. The results show that injections on `r7` lead to an exception in 75.3% of the cases.

6.2.2 Working with the control flow - Configuration 1

Configuration 1 includes the monitoring presented in [27] (*i.e.* with only the application independent code of the RTOS), with an extension of the control flow verification properties to 2 aspects:

- The possible states of the scheduler can be specialized according to the considered system call. This specialization reduces the acceptable states and increases the detection rate.
- The control flow check is also refined: in addition to checking the input/output parameters of functions it also verifies that the caller of the function is valid.

The Table 3 shows the results of the online monitoring error detection for this first configuration.

	r7	sp	lr	pc	Other registers	All
No impact	0.2%	0.1%	6.9%	10.5%	0.4%	0.6 %
Exception	8.5%	7.4%	13.9%	15.4%	35.6%	14.3%
Time out	0.0%	62.9%	20.0%	42.0%	5.6%	24.0%
SDC	73.3%	61.4%	84.3%	76.2%	73.3%	**75.2%**
	1,843	1,266	12,599	13,087	46,085	74880

TABLE 3
Fault injection campaign on Configuration 1: Monitor detection rate as a function of register type.

Only the kernel of the RTOS has been instrumented and this makes it possible to use the monitors for any application (*i.e.* no need to update the FPGA configuration). We can notice that 75.2 % of SDCs are detected directly which is a good trade-off considering the low hardware overhead of this solution (the rate was 48.4 % in the initial version without any control flow).

As seen in the previous section, some faults are already detected by the hardware (exceptions, timeouts) and the detection is complementary to the runtime verification mechanism. It can be noticed that faults that lead to a timeout are detected much more quickly with monitors, allowing for faster recovery. If a monitor detects an error during the fault injection phase, an interrupt is raised, but no recovery mechanism is set up. The program resumes its execution, and we can determine if a hardware mechanism later detects it as well. Thus, even if the goal is to detect SDCs, detecting non-silent faults is still useful because it allows the latency of detection to be reduced.

The runtime verification engine also detects faults that have no impact on the execution. This happens because an error has actually occurred, even if it has no impact on the scheduling. This type of detection should never be interpreted as a false positive.

At the binary level, the code instrumentation leads to an increase of 9.6 % of the size of the kernel code (memory overhead from 6436 to 7056 bytes). This instrumentation is very lightweight and compares very favorably to software-only verification techniques, such as [2].

Focusing more specifically on the SDCs, we can see that `lr`-related SDCs are very well detected (84.3 %) because the control flow is checked at both the beginning and at the end of each function. The detection rate for `pc` is lower (76.2 %). This register is modified after every instruction. If a bit flip in `pc` results in a sufficient offset, it will be detected by the control flow monitoring. However, an error that only causes a jump within a basic block may result in either re-execution or non-execution of few instructions. These alterations cannot be detected by the technique presented in this paper and are the result of a trade-off between execution performance and detection granularity.

The results for the stack register `sp` and the stack frame pointer `r7` are slightly below average: 61.4 % and 73.3 % of detected SDCs, respectively. The stack is used both for the control flow to save function return addresses of non-leaf functions (`lr` register backup) and for local variables. One solution to improve the robustness may be to limit the use of the stack especially for local variables.

6.2.3 Monitoring Application Dependent Kernel Code and Data - Configuration 2

Trampoline is a static RTOS, meaning that kernel data structures are tailored and defined at compile time. To fully exploit this approach, part of the kernel source code is generated from a static description of the application layer. Thus, we can distinguish two parts in the kernel code: the core kernel, which is constant on all systems, and the application-specific part, which is generated specifically for each system.

In configuration 1, we have focused on the monitoring of the core kernel. In configuration 2, in order to improve the detection rate of SDCs, the application-specific part of the kernel is taken into account. This includes descriptors of objects handled by the kernel, such as tasks or resources, and internal kernel data structures, such as the ready list, whose size is tailored according to static system requirements. Among the new properties are a property that ensures that the state of a task can only be set to RUNNING in the `run_elected` function, or a property that ensures that the size of the list of ready tasks can only be increased in the `put_preempted_proc` and `put_new_proc` functions. In the end, the number of ptLTL properties increases from 100 to 251.

Note that these monitors do not use any additional instrumentation: the binary is the same for both configurations. In this approach, monitors associated with application-specific kernel structures are also application-specific. Thus, they are automatically synthesized from the static description of the application layer along with the application-specific part of the kernel source code.

	r7	sp	lr	pc	Other registers	All
No impact	0.6%	0.1%	0.6%	10.8%	0.6%	0.8%
Exception	9.9%	8.5%	3.3%	13.6%	39.6%	15.4%
Time out	0.3%	62.9%	12.6%	80.6%	5.6%	24.1%
SDC	81.3%	64.1%	90.9%	81.8%	85.0%	**84.8%**
	2,043	1,322	13,578	14,047	53,421	84,411

TABLE 4
Fault injection on configuration 2: Monitor detection rate when application dependent kernel code is included.

Table 4 shows the detection rate of this new configuration. Since the number of properties has increased, the improvement affects all types of registers. Compared to configuration 1 (Table 3), the overall percentage of SDC detection is increased by almost 10 points, from 75.2 % to 84.8 %. These values are directly comparable because, since configuration 2 requires no additional instrumentation, so the binary code of the system is the same in both cases, as is the fault space.

6.3 Detailed analysis by register and areas for improvement

So far, when analyzing the results of fault injection campaigns, the focus has been on registers that control execution, while general-purpose registers have been aggregated into the "other registers" category. In table 5, the results for configuration 2 are given by registers for registers `r0` through `r5`. Registers `r6` and `r8-r12` are not used by the compiler for the kernel code, so no SDCs are associated with them.

The fault injections targeting r3 have a significant impact: they are responsible for almost half of the SDCs. One way to improve this might be to create monitors dedicated to faults triggered via r3.

	SDC	SDC detected	detection rate	remaining SDC
r0	8079	6144	76,0%	1935
r1	7995	7612	95,2%	383
r2	13592	11424	84,0%	2168
r3	31662	26712	84,4%	4950
r4	1410	1382	98,0%	28
r5	147	147	100,0%	0
			total	9464

TABLE 5
Detection of SDCs related to GPR r0 to r5 for configuration 2.

In the following, and based on the results already presented above, two new configurations are evaluated. In configuration 3, the goal is to limit the impact of faults injected through sp by modifying the use of the stack. Indeed, table 4 shows that the detection rate of faults injected via this register is 20 % lower than the average. In configuration 4, the goal is to limit the impact of faults injected through register r3.

These two new configurations require changes to the kernel source code. Thus, the fault space targeted by the fault injection campaign is different. In addition, configuration 4 uses new monitors. The comparison of the results obtained between these new configurations and configurations 1 and 2 is therefore not direct.

6.3.1 Limiting the stack usage - Configuration 3

To reduce the stack usage of the kernel, we have modified all the RTOS functions to convert local variables to global ones (type `static volatile`). This is made possible because the kernel is not reentrant. This modification is software-only, so the monitors are the same as configuration 2. However, the kernel code is modified and so is the fault space. The application trace totals L_{gt} =17 837 assembly instructions, an increase of 29.4 % over the previous version. The results of the fault injection campaign on configuration 3 are shown in Table 6.

	r7	sp	lr	pc	Other registers	All
No impact	0.7%	0.1%	0.5%	14.8%	0.6%	0.8%
Exception	7.5%	7.1%	2.8%	13.0%	27.1%	14.4%
Time out	1.1%	5.0%	53.2%	9.7%	3.1%	35.5%
SDC	89.2%	71.9%	87.9%	86.0%	91.6%	**90.2%**
	2652	1581	22,148	25,707	145,511	197,599

TABLE 6
Fault injection on configuration 3: Monitor detection rate with stack usage limitation feature.

With this configuration, the detection rate of SDCs is 90.2 %, which is an improvement over configurations 1 and 2. However, this figure must be put into perspective with the significant increase in the size of the binary code and, consequently, the size of the fault space. To illustrate this growth between configuration 2 and 3, the number of SDCs triggered by the fault injection campaign increased from 99,573 to 219,129. In practice, this means that even though the detection rate has increased (from 84.8 % to 90.2 %), the number of undetected SDCs has also increased (from 15 562 to 21 530). We observe that this configuration 3 leads to mitigated results whose interpretation is not obvious. We reach a limit of the approach.

6.3.2 Enhanced protection of kernel structure tpl_kern (r3) - Configuration 4

The tpl_kern data structure is important because it contains 3 bits that control scheduling behavior. Therefore, a more detailed analysis was performed on this data structure. We found the location in the binary code where the tpl_kern structure is accessed and intersected with the addresses of fault injections that lead to undetected SDCs. The analysis of these 124 different addresses shows that the SDC is always associated with the processing of a read operation. For instance:

```
123e: ldr   r3, [pc, #36] ; r3 <- @tpl_kern
1240: ldrb  r3, [r3, #28] ; r3 <- need_schedule
1242: cmp   r3, #0        ; re-schedule?
1244: beq.n 124a          ; branch if no ..
```

In this code snippet, the address of the structure tpl_kern is loaded into r3 (instruction address is 123e)[5] Then, r3 is loaded with the value of the need_schedule field which is at offset 28 of the tpl_kern structure. The last two instructions allow to select a branch according to the value of need_schedule. A fault injection in r3 at any of the first 3 instructions of the snippet may yield a wrong outcome for the comparison, resulting in the selection of the wrong branch and thus a false scheduling decision at the application level. This fault remains invisible to the monitors because the value of need_schedule in memory is not altered.

In configuration 4, we take this kind of situation into account, if possible, by instrumenting each conditional branch. It is then sufficient for the monitor to check, when the conditional block is reached, that the structure is in a coherent state, *i.e.* that the field tpl_kern that allows to establish the condition is correctly positioned. However, this does not apply if there is dynamic data that is not stored in the data structures (located in the FPGA). This happens for example when using function input parameters.

The size of the golden trace of the application is slightly increased to L_{gt} =13 839 % instructions (+0.41 % compared to configuration 1 and 2) due to the code instrumentation. The results of the detection rate using configuration 4 are reported in Table 7. Note that the first column now shows r3 instead of r7. The column “other registers” is the same as in the previous experiments and refers to registers r0-r6 and r8-r12.

	r3	sp	lr	pc	Other registers	All
no impact	5,0%	0.1%	0.6%	11.6%	0.6%	0.8%
Exception	41,3%	8.8%	3.5%	13.2%	39.5%	15.4%
Time out	6,7%	63.5%	13.8%	26.4%	8.7%	18.8%
SDC	84,7%	60.5%	87.9%	82.6%	85.3%	**84.6%**
	26903	1,190	12,760	14,426	54,202	84,655

TABLE 7
Fault injection on configuration 4: Monitor detection rate with enhanced protection of the tpl_kern structure.

5. On ARM architecture (here arm-v7m), the statically defined addresses used in the code are stored just after the function code. A fault injection on pc at this point will result in both an incorrect pointer to the structure and a change in control flow.

The results are very similar to those obtained with configuration 2. As for configuration 3, the fault space is larger due to the modification of the source code and the overall results are mitigated.

The number of SDCs detected on `r3` increases from 26 712 in configuration 2 to 26 903 here, which is 84.7 % of the total number of SDCs without runtime verification. However, if we consider all the registers, the number of possible SDCs increases with this new instrumentation and is not compensated by the higher number of detections, causing the detection rate to drop very slightly from 84.8 % to 84.6 %.

In conclusion, as with configuration 3, the instrumentation overhead and the modification of the fault space are not favorable compared to the benefit in detection rate.

6.4 Resource Consumption Analysis

Configurations 1 and 2 use the same binary, but configuration 2 has more monitors. Configuration 2 and 3 have different binaries, but use the same set of monitors. Finally, configuration 4 is different in both binary and monitor set. The implementation costs in terms of FPGA resources (4-LUT and flip-flops) of these different configurations are given in table 8 along with the golden trace size (instruction count), which is an indicator of the instrumentation overhead. Note: To harden the verification unit, the FPGA synthesis is performed using the Triplication Strategy (TMR) offered by the synthesis tool.

	L_{gt} (inst.)	ptLTL formulas	hardware resources	
			`4-LUT`	`flip-flop`
config. 1	13782	100	2456	2794
config. 2		253	6907	7800
config. 3	17837			
config. 4	13839	264	7327	7964

TABLE 8
Characteristics of the studied configurations. The hardware resources include the extra cost related to the TMR.

6.5 Lessons learnt from fault injection campaigns

In configuration 1, the addition of control flow monitors at the function level resulted in a doubling of the number of monitored properties (100 versus 49 in the original injection), but also a significant improvement in the SDC detection rate.

In configuration 2, the addition of monitors that target the application-specific part of the kernel further increases the number of monitored properties. Among these properties are some target objects that are managed by the kernel. They must therefore be instantiated multiple times: once for each instance of the object in the system. In the case of the reference application used for the fault injection campaign it includes 6 tasks and 3 synchronization objects (2 *events* and 1 *resource*), and the number of properties is 253 (*vs.* 100 in configuration 2). The consumption of FPGA resources by monitors associated with kernel objects is relatively small: each *task* requires about 500 flip-flops and 426 4-LUTs, and each *resource* requires 370 flip-flops and 322 4-LUTs with TMR enabled. Our reference application uses a total of 6907 4-LUTs and 7800 flip-flops (with TMR enabled).

Configuration 4, which uses additional monitors, is barely more consuming than configurations 2 and 3. From an error detection point of view the results obtained for configurations 3 and 4 are mixed: they improve the detection rate but also lead to an increase in the number of undetected SDCs. As it is, configuration 2 is probably close to the best performance achievable with the approach developed here.

7 Conclusion

In this paper, we have evaluated hardware accelerated runtime verification to detect silent data corruptions resulting from runtime physical faults. We have applied the proposed technique to monitor the execution of Trampoline, a small RTOS for deeply embedded systems.

The system is built on top of a SmartFusion2 SoPC: The software under verification runs on the micro-controller while the verification unit is implemented as hardware blocks on the FPGA and runs in parallel. The software is instrumented to provide information to the verification unit.

On the software side the instrumentation causes an overhead between 15.2 % and 38.1 % on the execution time of services and an increase of less than 10 % of the size of the code of the kernel. On the hardware side the verification unit consumes less than 13 % of the FPGA resources leaving the possibility to use the remaining resources for accelerators.

Four different configurations of monitors and instrumentation have been evaluated through software-implemented fault injection campaigns. The configuration that appears to be the most balanced at the end of the evaluation phase detects 84.8 % of SDCs. We believe this detection rate is close to what can be achieved with this approach without otherwise degrading other performance metrics. Overall, the technique provides partial SDC detection for a very reasonable overhead in terms of time, memory and FPGA resources.

References

[1] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline - an opensource implementation of the osek/vdx rtos specification," in *11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'06)*, september 2006.

[2] S. Cotard, S. Faucou, J. L. Béchennec, A. Queudet, and Y. Trinquet, "A data flow monitoring service based on runtime verification for autosar," in *IEEE International Conference on Embedded Software and Systems*, June 2012, pp. 1508–1515.

[3] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Copilot: monitoring embedded systems," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, p. 235–255, Dec. 2013.

[4] K. Morin-Allory and D. Borrione, "Proven correct monitors from PSL specifications," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006, pp. 1246–1251.

[5] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of PSL properties," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 1, p. 4, 2008.

[6] H. Lu and A. Forin, "The design and implementation of p2v, an architecture for zero-overhead online verification of software programs," Technical Report MSR-TR-2007–99, Microsoft Research, Tech. Rep., 2007.

[7] T. Reinbacher, J. Brauer, D. Schachinger, A. Steininger, and S. Kowalewski, "Automated test-trace inspection for microcontroller binary code," in *International Conference on Runtime Verification*, ser. RV'11, 2012, pp. 239–244.

[8] M. A. Wahab, P. Cotret, M. Nasr Allah, G. Hiet, V. Lapotre, and G. Gogniat, "ARMHEx: A hardware extension for DIFT on ARM-based SoCs," in *International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017.

[9] D. Solet, S. Pillement, J.-L. Béchennec, M. Briday, and S. Faucou, "Hw-based architecture for runtime verification of embedded software on SoPC systems," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2018.

[10] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[11] K. Sen and G. Rosu, "Generating optimal monitors for extended regular expressions," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 226–245, 2003.

[12] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[13] K. Havelund and G. Rosu, "Synthesizing monitors for safety properties," in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '02, 2002, pp. 342–356.

[14] T. Reinbacher, K. Y. Rozier, and J. Schumann, *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 8413, p. 357–372. [Online]. Available: http://link.springer.com/10.1007/978-3-642-54862-8_24

[15] *Property Specification Language : Reference Manual*, Accellera, 2004.

[16] M. Rodriguez, J. C. Fabre, and J. Arlat, "Formal specification for building robust real-time microkernels," in *21st IEEE Real-Time Systems Symposium, RTSS 2000*, 2000, pp. 119–128.

[17] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann, "dosek: the design and implementation of a dependability-oriented static embedded kernel," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2015*, 2015, pp. 259–270.

[18] AUTOSAR, "Specification of Operating System," part of AUTOSAR Release 4.2.2.

[19] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits," *Integration*, vol. 71, pp. 154–163, Mar. 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016792601930402X

[20] A. Aponte-Moreno, F. Restrepo-Calle, and C. Pedraza, "Reliability evaluation of risc-v and arm microprocessors through a new fault injection tool," in *2021 IEEE 22nd Latin American Test Symposium (LATS)*, 2021, pp. 1–6.

[21] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Portland, OR, USA: IEEE, Jun. 2019, pp. 26–38.

[22] D. Skarin, R. Barbosa, and J. Karlsson, "Goofi-2: A tool for experimental dependability assessment," in *IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 557–562.

[23] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Denver, CO, USA: IEEE, Jun. 2017, pp. 97–108.

[24] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," in *2015 11th European Dependable Computing Conference (EDCC)*, 2015, pp. 245–255.

[25] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A comparison of inject-on-read and inject-on-write in isa-level fault injection," in *2015 11th European Dependable Computing Conference (EDCC)*, Sept 2015, pp. 178–189.

[26] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *International Conference on Computer Safety, Reliability, and Security - Volume 8153*, 2013, pp. 265–276.

[27] D. Solet, M. Briday, J.-L. Béchennec, S. Faucou, and S. Pillement, "Hardware Runtime Verification of a RTOS Kernel: Evaluation Using Fault Injection," in *14th European Dependable Computing Conference (EDCC)*, Iasi, Romania, Sep. 2018.

Dimitry Solet is teacher in the embedded system department of the ESEO engineering school of Angers (France) since 2020. He has a *doctorat* (PhD) in Computer Sciences from Université de Nantes (2020).

Jean-Luc Béchennec is a full-time researcher at CNRS (France) in the Research Laboratory of Digital Sciences of Nantes (LS2N, UMR 6004). He earned a PhD degree in University of Paris VI in 1989. He works mainly in the fields of system and RTOS implementation, runtime monitoring and modeling and verification by using formal methods.

Mikaël Briday Mikaël Briday is an Associate Professor at École Centrale de Nantes (France). His research activity is carried out at LS2N in Nantes (LS2N, UMR 6004). It focuses on real-time embedded systems, with an emphasis on the robustness of critical embedded systems, RTOS implementation and energy management in the IoT. He received its Ph.D. in 2004 for his work on validation of real-time systems through simulation.

Sébastien Faucou is Maître de Conférences at Nantes Université (France). He has a DEA (MSc.) in Automatic Control and Applied Informatics from École Centrale Nantes (1999) and a *doctorat* (PhD) in Computer Sciences from Université de Nantes (2002). He is head of the Real-Time Systems group of the *Laboratoire des Sciences du Numérique de Nantes* (LS2N, UMR 6004). His work deals with the design, verification, and optimization of embedded software systems.

Sébastien Pillement is full Professor in Polytech'Nantes, France since 2012. He was associate professor at the University of Rennes 1, during 13 years. He is currently a research member of the ASIC Research Team of the IETR Lab. (Research Institute in Electronic and digital technologies). His research interests include dynamically reconfigurable architectures, system on chips, design methodology and NoC (Network on Chip) based circuits. He focuses his research on designing flexible and efficient architectures managed in real-time and on the use of these architectures to increase the reliability of systems. He is the author or coauthor of about 100 journals and conference papers.