Soft-Error Classification and Impact Analysis on Real-Time Operating Systems

N. Ignat, B. Nicolescu, Y. Savaria, G. Nicolescu
Ecole Polytechnique de Montréal, P.O. Box 6079, Station Centre-Ville, Montréal, Canada

Abstract
This paper investigates the sensitivity of real-time systems running applications under operating systems that are subject to soft-errors. We consider applications using different real-time operating system services: scheduling, time and memory management, intertask communication and synchronization. We report results of a detailed analysis regarding the impact of soft-errors on real-time operating systems cores, taking into account the application timing constraints. Our results show the extent to which soft-errors occurring in a real-time operating system’s kernel impact its reliability.

1. Introduction
Real-time systems must control and monitor physical processes in a timely manner [1]. Consequently, a real-time system must provide not only logically correct results, but also temporally correct results. If a result is logically correct, but it is obtained too early or too late, the system behavior may be incorrect, and the consequences may be catastrophic. In this context, for a real-time system, several non-functional requirements, such as timing, availability, reliability and safety are as important as the functional requirements.

A special category of real-time systems is hard real-time systems, in which it is imperative that tasks (related to application functionalities and to external events) meet their deadlines. If deadlines are not met, the system may endanger human lives, the environment and/or valuable assets. Traffic control, process control, medical life support, satellite and space stations, nuclear power stations management applications are only a few examples of hard real-time applications.

Due to tight constraints, the design of a real-time embedded system, and especially of a hard real-time system, tends to be delicate. Using a real-time operating system (RTOS) can simplify the design of a real-time application. An RTOS allows developers to split the application into separate tasks, of different priorities, and to use the RTOS intrinsic mechanisms to manage tasks, concurrency, memory, time, interrupts, etc. If these services do not work properly, the task execution order may be affected and some critical tasks could miss their deadlines and/or provide incorrect logical results.

Moreover, real-time embedded systems are subject to different parasitic phenomena induced by the environment (e.g. Single Event Upsets [2]). Such faults and the soft errors they cause are a growing concern of the microelectronics industry. This class of errors may cause scheduling dysfunctions that could lead to incorrect system behavior. For hard real-time systems, the RTOS main services must be robust if the system is to continue functioning even in the presence of faults.

In this context, various researchers are concerned with real-time systems robustness in presence of faults. Some of them propose diverse approaches for robust scheduling algorithms [11]-[13], others are focused on the task reexecution and/or replication [17] and finally, another group of researchers proposes additional application tasks in order to check the application tasks workspace memory [16].

As a general characteristic of existing works, they propose fault tolerant solutions only for the application level. However, faults in the OS that propagate to the application tasks may have critical consequences, defeating the proposed fault tolerant mechanisms. Such faults may be seen as unlikely, but in our experiments, it was found to be frequent enough to produce a significant system failure rate. Further investigations of causes for system failures and error propagation are required.

With respect to existing works, this paper analyses the impact of soft-errors on the different groups of MicroC RTOS [9] services. We also investigate error propagation when faults affect communication and synchronization mechanisms used in the studied real-time multitasking application.

The rest of the paper is organized as follows. Section 2 presents some previous research related to the evaluation of RTOS robustness. Section 3 gives an
overview of the experimental framework used in our research related to the adopted fault injection strategy, the studied RTOS main features and the real-time application characteristics. Section 4 presents a classification of errors. Experimental results and their analysis are proposed in Section 5. Finally, conclusions and some directions for future work are given in Section 6.

2. Related research

In the last decade, several researchers have studied fault-tolerant real-time scheduling algorithms for single processor systems [11]-[13]. These algorithms are used to improve processor utilization and to protect real-time systems against error effects, by detection and recovery from errors. Some examples of such algorithms are Fault-Tolerant Rate Monotonic Scheduling (FT-RMS) [10], Imprecise Computation Fault-Tolerant Rate Monotonic Scheduling (IC-FT-RMS) [11] and [12]. The main idea of these algorithms is to reserve sufficient time for error recovery in the tasks schedule. This time is used to detect errors in the application tasks and eventually re-execute the faulty tasks or their backups, with respect to all critical tasks deadlines.

Other researchers evaluated the robustness of real-time microkernels components (scheduling, intertask communication, etc.). In [5], fault injection with the FINE tool was used to study fault propagation in the UNIX operating system. Experimental results showed that most injected faults lead to system failures. In [6], a fault injection tool corrupting POSIX system calls parameters was proposed. The results showed that most of the POSIX functions failed when a system is subject to faults. In [14], [3] and [4], authors assess the robustness of the Chorus microkernel. Reported results were related not only to system crashes, but also to error propagation at the application level. The research presented in [14] investigates the impact of transient faults on the timing response of real-time applications. The authors study the consequences of faults injected in the Chorus’s scheduler code. Experimental results show that about 7% of injected faults are propagated to the application level, causing incorrect timing and logical results.

3. Experimental framework

3.1. MicroC: the selected RTOS

MicroC is a portable, scalable, multitasking, preemptive, deterministic, robust and reliable real-time kernel [9]. In July 2000, the Federal Aviation Administration (FAA) certified it for the use in safety-critical systems. We chose MicroC for our research because it is a very popular real-time kernel, used in hundreds of products, and it is well known in the academic world; its source code and documentation are accessible, clean and easy to understand.

In MicroC, a task is typically an infinite loop that executes a part of the user application. Each task has a priority, based on its importance and the specific features of the application. The MicroC scheduler is priority-based and preemptive.

Since it is a real-time operating system, MicroC kernel is deterministic, consequently, the scheduling time is always constant and it does not depend on the number of application tasks. In order to achieve this goal, MicroC uses dedicated data structures to keep information about tasks’ state. For example, the scheduler uses the OSRdyGrp and OSRdyTbl structures to determine the highest priority task that will run next.

Taking into account this functionality, soft-errors affecting services that perform operations in these structures may have major implications on the correct behavior of the MicroC kernel and implicitly, on the real-time application that it runs.

3.2. Fault injection environment

This section briefly describes the main features of the adopted fault injection mechanisms. This fault injection technique was initially proposed in [15]. To identify sensitive components of the studied real-time operating systems, faults were injected in the CPU registers during execution of the MicroC RTOS main services.

The adopted system architecture is simulated with an Instruction Set Simulator (ISS) [18] tool that targets the HC12 microcontroller from Motorola [19]. The fault injection tool uses temporal breakpoint features available in the ISS to inject faults by software means. Once a temporal breakpoint is reached, global execution is suspended and the ISS activates a Fault Injection Manager (FIM) that comprises three modules:

- Fault parameters generator – calculates when and where the fault will be injected. FIM can inject faults in CPU registers, cache memories or internal and external memories, but in our experiments, faults consist of single bit-flips only in the CPU registers while the main services of the MicroC kernel are active,
• **Fault tracer** – collects information about the currently executed task. At each task execution, it saves into a file the instants when the task starts and ends its execution as well as the tasks’ output results,

• **Results analyzer** – uses information provided by the fault tracer module in order to classify the fault consequences.

After the fault has been injected, the global execution is resumed. The injection process is depicted in Figure 1.

**Figure 1: Fault Injection process**

The main advantage of the adopted fault injection approach is that the fault injection manager acts as an independent module. It does not perturb the normal system execution to inject faults. Accordingly, the multitasking application and the MicroC kernel are not modified by the fault injection tool. After each injection, the target system is restarted in order to ensure independent experiments.

### 3.3. Characteristics of the benchmark program

In order to perform our experiments, we adapted a multitasking application provided by Labrosse, the MicroC’s author [20], to our system’s requirements. We choose this application because it is complex and it uses the most important services offered by MicroC real-time kernel. The studied application consists of 15 tasks that can be organized in 5 groups, according to the mechanisms through which they communicate and synchronize (see Figure 2):

- **Group1** – tasks T1, T2, T3 and T4 share memory blocks to store their results;
- **Group2** – tasks T5, T6 and T7 communicate by message queues. T5 (transmitter) sends the results of its computations into QM1 and QM2 message queues. T6 and T7 (receivers) read messages from T5, and use them in their functionalities;
- **Group3** – tasks T8, T9, T10 and T11 use event flags when tasks need to synchronize with multiple events;
- **Group4** – tasks T12 and T13 access a global variable protected by mutual exclusion semaphore (mutex);
- **Group5** – tasks T14 and T15 communicate by a mailbox that can store a single message. T14 sends a message periodically into a mailbox, while T15, the receiver, consumes the message and uses it in its future operations.

All tasks are considered critical (they must always complete execution before their deadlines and produce logically correct responses) and perform some useful computations. Before performing any fault injection experiment, we carefully studied and tested this real-time application in an environment without soft-errors to verify that all tasks meet their deadlines and produce correct output results.

**Figure 2: Tasks relationships in the studied application**

### 4. Experimental results

This section describes and analyses the obtained results to get evidence of soft-errors consequences in the case of a real-time application.

#### 4.1. Fault syndromes classification

Transient faults may cause several syndromes when the real-time kernel’s services are corrupted. These syndromes are classified as follows:

- **Effectless** – no visible effect on system functionality;


• Exception trigger – the program triggers some exception routine (e.g. illegal instruction, division by zero, etc.);
• System crash – the system stops functioning;
• Application failure – represents a class of faults with visible consequences on the application level.
  This class of faults can be subdivided as:
  ▪ Incorrect output results – one or more application tasks are able to provide results, but they are different from the expected ones;
  ▪ Real-time problem – one or more application tasks do not respect their real-time constraints;
  ▪ Task hang – system still works but one or more application tasks stop functioning.

4.2. Results analysis

To assess the MicroC kernel sensitivity to transient faults, we performed several fault injection campaigns. Faults were randomly injected in the CPU registers during execution of MicroC services. Note that, we did not consider the issue of injecting faults in the MicroC services that are used only for system initialization (e.g. the task creation service that is used only when the application tasks are created).

The impact of soft-errors according to the different groups of MicroC services is illustrated in Figure 3. The category axis (X) illustrates the classes of fault consequences, while the value axis (Y) shows their respective occurrence frequency. The different groups of MicroC services are depicted by a column bar. For instance, consequences of faults that affect services belonging to the time management group are illustrated by the column bar with vertical lines pattern (the 4th column in each class of fault consequences).

A remarkable feature of our results that is apparent from Figure 3 is that all groups of MicroC services share almost the same sensitivity profile.

As a general observation, for all groups of MicroC services, the predominant class of fault syndromes is the effect-less class. On average for all groups of services, about 42% of faults have no visible effects on the system behavior. Unlike the effect-less class syndrome, a very small fraction of faults (4%) trigger exception routines (illegal instruction or division by zero). System crashes represent 32% (average for all services groups) of faults. Regarding faults that propagate to the application tasks, about 21% of them led to application failures.

According to the type of fault consequence, the application failure class can be subdivided as illustrated in Figure 4. Generally, when faults propagate to the application level, tasks tend to miss their deadlines and to produce incorrect output results. Task hang was produced by 26% of faults in the application failure class.

Several valuable observations can be derived from the comparison of the results obtained in this paper with those presented in [15]. The sensitivity of the MicroC kernel appears to be application independent. Indeed, for a much less complex real-time application, similar results were obtained. Moreover, with respect to this paper, the study reported in [15] was performed on a single group of MicroC services: the scheduler and context switch modules. The results obtained in this paper show that not only the scheduler and context switch are sensitive to soft-errors, but also all MicroC services have almost the same sensitivity profile.

![Figure 3: Fault syndromes according to different groups of MicroC services](image-url)

![Figure 4: Classes of faults that propagate to the application level (the application failure class)](image-url)
We also investigated consequences of faults that affect the communication and synchronization mechanisms used in our application as illustrated in Figure 5. As shown in this figure, for the considered application, faults that affect the mutex, flag group and semaphore communication and synchronization mechanisms tend to affect tasks that belong to other groups. For instance, when faults affect the mutex communication services, only 20% of faults have consequences on tasks that use the mutual exclusion (T12 and T13, see Figure 2). With the remaining 80%, the effects are propagated to other groups of tasks (for example, T5 that belongs to the group of tasks that communicate by message queues).

Concerning the message box and message queue communication mechanisms, we observed that most of the injected faults have consequences on the same group of tasks. Indeed, 75% of faults affect the group of tasks that communicates by message queues: T5, T6 and T7 (see Figure 2).

4.3. Discussion on fault tolerance in real-time systems

Our experiments have shown that real-time systems are very sensitive to soft-errors. Therefore, designers concerned with system robustness in presence of soft-errors should be aware of the different classes of fault syndromes specific to real-time systems. This sets requirements for new fault tolerance mechanisms.

In the specialized literature, to mitigate application failure fault syndromes, several researchers proposed robust scheduling algorithms to cope with temporal constraints [11]-[13]. Others propose introduction of additional tasks that periodically checks the system’s workspace memory, but they neglect the temporal aspects [16]. Finally, other works propose the reexecution or replication of the application tasks to enforce system robustness [17].

However, faults that propagate to the application level may have malicious consequences. As an illustrative example, some tasks may loose the link to their context. This happens when the stack pointer (SP) is corrupted while the context switch function loads the context of the ready to run task. The consequence of this type of fault can be permanent; the task will permanently compute with an erroneous context after fault injection. Another example for faults with critical consequences on the application task corresponds to situations when a task is reexecuted within the same period and at each execution, its real-time requirements are respected, as illustrated in Figure 6.

![Figure 6: Double execution of a task within the same period](image)

In this situation, for the task in Figure 6, the real-time condition $C \leq D \leq P$ is respected, but the fault may introduce a significant delay for all other tasks. Such faults that result in scheduling dysfunctions may
bypass even the robust scheduling algorithms proposed in state-of-the-art works. In our experiments, we observed that about 40% of faults that affect the scheduler led to tasks reexecution within the same period.

Moreover, it was found that more than 25% of faults affect several application tasks. Obviously, proposed fault tolerance techniques can represent feasible solutions, but they cannot guarantee that tasks respect their deadlines. However, relying on fault tolerance mechanisms to obtain dependability often means accepting some performance degradation that makes it more difficult to respect the system real-time constraints.

5. Conclusions and future work

Nowadays, safety-critical applications are often based on real-time operating systems. These systems are subject to faults that affect both the correctness of logical results and the timing of tasks response.

In this paper, we reported a detailed analysis of soft-errors impact on the key services of MicroC, taking into account the application timing constraints. Our results show that soft-errors occurring in a real-time operating system’s kernel have a major impact on the system’s behavior. Moreover, it was found that all groups of MicroC services have the same sensitivity profile. We also investigated consequences of faults that affect the communication mechanisms used in our application. According to our results, when faults affect the mutex, flag group and semaphore services, their consequences have a tendency to propagate to tasks not necessarily using these services.

Investigations on faults that propagate to the application level proved that these faults may have critical consequences that can bypass fault tolerance mechanisms proposed in the specialized literature. Furthermore, more than 25% of injected (single) faults affect more than one application tasks. This demands new efforts for error detection/correction mechanisms.

In our future research, we plan to implement robust RTOS that includes mechanisms for error detection and correction. These mechanisms should notably be applicable to the RTOS kernel, in order to improve its robustness.

6. References

[18] “Motorola HC12 CPU awareness and true-time simulation”, Metrowerks Corp., 2004