A Reliable and Predictable Scratchpad-Centric OS for Multi-Core Embedded Systems

Rohan Tabish*, Renato Mancuso*, Saud Wasly**, Sujit S. Phatak+, Rodolfo Pellizzoni** and Marco Caccamo*
*University of Illinois at Urbana-Champaign, USA, {rtabish, rmancus2, mcaccamo}@illinois.edu
**University of Waterloo, Canada, {swasly, rpellizz}@uwaterloo.ca
+ Hitachi America, Ltd., sujit.phatak@hitachi-automotive.us

***Abstract*—The reliable use of multi-core platforms for designing safety-critical systems still represents an open challenge. Recently, the FAA [1] has formally expressed its concern towards the use of multi-core systems in avionics. The sharing of hardware resources introduces non-trivial timing dependencies between logically independent components (e.g. cores); additionally, the increase in size of circuitry, memory resources, and transistor density makes these platforms more susceptible to transient memory (soft) errors.**

This work addresses the problem of memory soft errors and their recovery at an OS/platform level on commercial multi-core systems. Proposed strategy considers the schedulability impact of recovery procedures on hard real-time workloads. Finally, the implementation of a SPM-centric OS with the proposed OS-level strategies was performed by using a commercially available multi-core platform. The design has been validated and evaluated using a combination of synthetic and realistic (EEMBC) benchmarks.

I. Introduction

The high complexity of modern embedded and safety-critical applications has resulted in a substantial increase in the demand for computational power. Moreover, the traditional strict constraints in terms of size, weight and power consumption still apply. Multi-core processors represent the industry response to such an increasing demand for computational power required by embedded applications. Additionally, the massive transition of hardware vendors to multi-core platforms has drastically impacted the availability of single-core systems. Unfortunately, the reliable use of multi-core platforms for building safety-critical systems still represents an open challenge. Recently, the FAA [1] has formally expressed its concern towards the use of multi-core systems in avionics.

The fundamental issue resides in the increased complexity of multi-core hardware platforms. The impact of such an increase in complexity is twofold. On one hand, non-trivial timing dependencies are introduced between logically independent components (e.g. cores) due to a fundamental sharing of hardware resources. On the other hand, the substantial increase in size of circuitry, memory resources, and transistor density makes modern platforms more susceptible to transient memory (soft) errors.

The problem of hardware resource contention in multi-core systems has been extensively studied by the research community, and several different works [2, 3, 4, 5, 6, 7] have been proposed to mitigate its effects on application timing. Similarly, a body of work has addressed how to recover from soft errors via hardware-only or software-aided techniques [8, 9]. However, predictability and error handling for safety-critical, real-time applications have evolved on parallel tracks. This work represents an attempt in bridging the gap between predictability and reliability, in that it proposes a unified technology to (i) achieve strict timing determinism for the execution of real-time applications on multiple cores; and (ii) a comprehensive set of software techniques to recover from detectable soft errors.

The ability to recover from soft errors and effectively to extend the Mean Time To Failure (MTTF) is particularly relevant when considering safety-critical systems. This is because real time embedded devices are often deployed in hostile environments such as production plants, aircraft, and satellites. In such environments, extended exposure to various kind of radiations, such as alpha particles, high and low energy cosmic rays, as well as strong electromagnetic fields, can increase the probability of temporary "bit flips", i.e. soft errors, in the circuitry. The rate at which soft errors occur is called the Soft Error Rate (SER). The commonly used unit of measure for SER is the Failure In Time (FIT). One FIT is equivalent to one failure in 10^9 device hours.

It has been estimated that the FIT/bit of SRAM memories is about 0.001, i.e. one soft error per-bit every 10^{12} hours of operation [10]. The probability of failure for each bit follows an exponential failure distribution. Let us assume that failure events are independent from each other. Therefore, a first-order approximation for the probability of (at least) one failure in the entire memory subsystem can be obtained by multiplying the FIT/bit by the total number of bits in the system. As a reference, the platform considered in our evaluation features about $1\ MB = 8 \cdot 10^6$ bits of SRAM memory including main memory and local SRAM, also known as scratchpad memories (SPM). This means that each SoC will experience a single-bit soft error every $8 \cdot 10^{-6}$ hours, i.e. about 0.0002 errors per day. According to statistics about the worldwide population of automotive vehicles [11], about half a billion cars were present in 2010, with numbers expected to exceed the billion by 2020. The year 2020 also corresponds to the expected commercialization of self-driving technology, which will arguably determine an increase in the complexity of the computing infrastructure of automotive vehicles. Let us consider the current FIT for SRAM memories, and conservatively suppose that the worldwide car population does not exceed 0.5 billion. Even assuming an average daily operation time of 5%, about 5000 vehicles per day will be affect by a soft error, with potentially catastrophic consequences.

In this work, we consider the case of detectable SRAM memory errors and propose a set of OS-level strategies to recover affected real-time applications into a full operational state, without violating their timing constraints. The proposed

978-1-5090-5269-1/17 $31.00 © 2017 IEEE

strategies are designed and implemented on top of our previously proposed SPM-centric OS [12]. SPM-centric OS refers to a scratchpad (SPM) oriented OS design where hard real-time tasks are scheduled and executed following a three-phase structure. In the first phase, a task to be executed is *loaded* from main (SRAM) memory into the local SPM of a CPU. Next, the task is locally executed by the CPU. Finally, the state that needs to be preserved across task releases is *unloaded* back into SRAM. Load/unload operations are performed using a DMA, hence they can progress in parallel with the execution phase of a different task.

Hereby, we describe how the SPM-centric OS has been extended to recover from bit errors, in both SRAM and SPM, that are detected but not corrected by the hardware logic via error-correcting code (ECC). The strategy that we follow largely leverages the existing redundancy in the employed multi-phase task model. A minimum amount of additional redundancy is introduced to protect data that do not exist as multiple copies in the original scheme. Hence, we make the following contributions:

1) We propose a set of strategies that can be used in systems that implement a multi-phase task model to recover from detectable bit errors;
2) We provide a schedulability analysis to take into account the overhead introduced by the proposed recovery mechanisms;
3) We extend the implementation of our SPM-centric OS with the proposed OS-level strategies using a commercially available multi-core platform. The design has been validated and evaluated using a combination of synthetic and realistic (EEMBC) benchmarks.

It should be noted that our proposed techniques presented in this work are able to improve the error correction capabilities of modern SoCs' ECC modules by working on top of them, rather than re-implementing their functionality from scratch. Since the implementation of HW ECC modules greatly varies in capabilities, our proposed techniques work with different kinds of ECC modules. For instance, in the case of parity-based ECC modules that are only able to detect (no correction) a single-bit error, our techniques can be used to perform correction using the detection capabilities provided by the hardware. Whereas in case of the ECC modules where the hardware provides single-bit error correction and double-bit error detection (no correction for double-bit errors), also known as SEC-DED, our techniques rely on the hardware for single-bit error correction. However, they extend the error correction capabilities to double-bit errors by relying on hardware double-bit error detection mechanism and redundant copies of application memory.

The rest of the paper is organized as follows. In Section II we revise the related work. Section III discusses the system model and assumptions. Next, we describe the proposed OS-level recovery strategies in Section IV. In Section V, we provide a schedulability analysis for real-time tasks subject to faults and recovery procedures. Additional details about our implementation are provided in Section VI. The proposed methodology and implementation is evaluated in Section VII. Finally, Section VIII concludes the paper.

II. Related Work

A consistent number of works have investigated both hardware modifications [13, 14, 15] and OS-level techniques [16, 2, 17, 12] to achieve predictability in spite of hardware resource sharing. On the other end, modifications to scheduling algorithms and schedulability analysis techniques to take into account the effect of faults has been largely investigated in [18, 19, 20, 21, 22].

In terms of timing predictability, it has been demonstrated that promising results can be achieved with hardware modifications [13, 14, 15]. In this work, however, we focus on OS-level modifications to achieve predictability on commercial multi-core system, requiring no hardware modifications. The benefits of software-only approaches mainly concern cost-effectiveness and time-to-market minimization, since their adoption does not involve hardware manufacturing processes.

Several works have proposed OS-level modifications that can improve timing determinism of commercial multi-core systems. A number of these works have focused on the implementation of scheduling techniques for multi-core systems [23, 24, 25, 17]. Many of these works have used the popular Linux-based LITMUS^{RT} to test proof-of-concept implementation of scheduling algorithms [16].

A second group of works have undertaken the challenge of enforcing OS-level management of hardware resources to reduce inter-core interference and to achieve timing determinism. The MC^2 framework initially proposed in [17] was integrated with hardware management techniques to control inter-core interference through shared resource partitioning [4] and the trade-off between strict partitioning and data sharing was evaluated in [26].

The work in [2, 3] proposes the idea of "Single Core Equivalence" (SCE): a framework of OS-level techniques to achieve predictable task execution on cache-based multi-core systems. Under SCE, strict hardware resource management is enforced to mitigate inter-core interference at the level of shared caches, memory controller, DRAM banks and I/O devices. When SCE is deployed, the performance overhead arising from resource sharing can be bounded and analyzed.

On scratchpad-based architectures, a number of memory allocation strategies targeting real-time systems have been proposed [5, 6, 7]. Alongside, scheduling analysis for methodologies that involve scratchpad management have been investigated in [6, 7, 27]. In this context, our previous work [12] has discussed the design and implementation of a scratchpad-centric OS: i.e. an OS that uses scratchpad management with DMA and CPU pipelining to achieve predictability on commercial multi-core systems *by design*. The SPM-OS design shares a number of similarities with the Acquisition Execution Restitution (AER) task model [28], whose scheduling properties have been recently studied in [29].

Addressing fault recovery in real-time systems represents an open challenge. Many theoretical results have been proposed to extend schedulability analysis accounting for task-level faults and restarts. The works in [18, 19, 21] analyze systems where a minimum inter-arrival between faults can be established, while an analysis that considers bursts of faults is proposed in [20]. On the other end, system-level studies have considered the implementation of fault recovery techniques without taking hard real-time constraints into account. For instance, the works in [30, 31] have used full system reboot as a recovery mechanism. Similarly, a proactive approach for memory fault avoidance and a software-based strategy to recover from storage faults were proposed in [32] and [33] respectively.

Compared to related work [20, 21, 22], the proposed recovery strategy jointly addresses not only application-level fault

recovery, but also the memory contention problem suffered by real-time applications running on multicore. While [20, 21, 22] have discussed the impact of recovery on schedulability analysis, these works do not attempt to model the multi-core memory interference at the level of shared memory and bus; hence, their schedulability analysis would have to be extended if the recovery strategy was deployed on multi-core platforms. It follows that the contribution of proposed work is twofold: (i) unlike the mentioned works [20, 21, 22] we propose an implementation on commercial hardware by extending an existing RTOS; (ii) while the literature in fault-tolerant scheduling has essentially targeted single-core systems, we propose and implement a real-time recovery strategy for multi-core platforms. The strategy is based on task reloading, it integrates seamlessly with the adopted memory management scheme, and its impact on schedulability analysis can be accounted and evaluated.

To the best of our knowledge, this work is one of the first to address fault recovery at an OS/platform level on commercial multi-core systems, while also considering the design, implementation and impact of recovery procedures on hard real-time workload. In fact, what sets this work apart from the aforementioned literature is the establishment of a complete OS-level framework to achieve both (i) strong time determinism for hard real-time tasks; and (ii) a wide range of capabilities to recover from detectable memory errors.

III. System Model and Assumptions

In this section, we summarize the task model and the hardware assumptions that we make for incorporating the recovery mechanisms into our SPM-centric OS. This section discusses assumption about the hardware and the task models.

a) Scratchpad Memories: We assume that each core in the multi-core environment has scratchpad memories. The size of scratchpad memories is big enough to fully contain the footprint of at least two copies of the biggest tasks in the considered task set.

b) DMA Engines: In order to avoid the CPU stall during the load/unload phase of a task and allow parallel execution, we assume that the hardware provides a direct memory access engine (DMA). Together with the previous assumption on the size of the scratchpad memory along with the assumption of having a DMA module, we allow parallel loading/unloading of the tasks from one partition and their execution from the other.

c) Error Detection On SRAM, Flash and SPM: We assume that hardware error detection logic is available for different kinds of memories, such as SPM, SRAM and Flash. For the purpose of our evaluation, we considered error detection capabilities of the Freescale MPC5777M processor. Our platform implements Hsaio codes that provide single bit error correction and double bit error detection (SEC-DED). Hsiao Code [34] for correction and detection is popular in modern embedded platforms.

Although for our implementation we use double bit error detection, our approach works with any kind of error detection logic, as long as it is possible for the OS to determine the location of the error in physical memory.

d) I/O Subsystem: We assume that there is a separate core to handle and manage the data from the I/O devices, we name this core as the I/O core. The I/O core has a separate bus using which it handles all the peripherals such that they do not interfere with the normal operation of the application cores. Figure 1 shows an overview of the considered architecture.

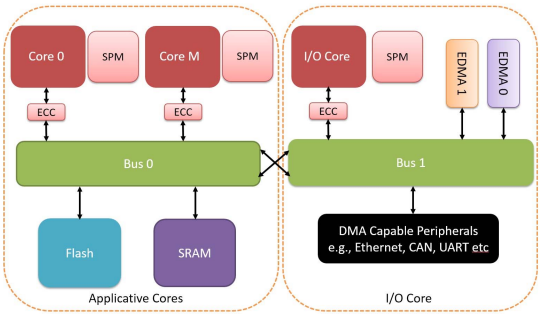

Fig. 1. Multi-core architecture satisfying our hardware assumptions

TABLE I
Task's Parameters

Term	Definition
τ_i	a task in the system
$\tau_i.T$	task's MIT or period (if task is periodic)
$\tau_i.c$	task's execution time including all overheads
σ	TDMA slot size for the DMA operation
ρ	ISR recovery overhead on applicative core

e) Task Model: For the proposed design, we consider a partitioned and fixed priority scheduling policy; additionally, each core has a set Γ of N sporadic tasks, $\{\tau_1, \ldots, \tau_N\}$, each with different priority whereby τ_1 has the highest priority and τ_N has the lowest priority. The deadline of each task is assumed to be less than or equal to its Minimum Inter-arrival Time (MIT). Table I summarizes the notation used for task parameters. As discussed in Section IV, tasks follow a three-phases model. Hence, to satisfy temporal constraints, the last phase (unload) of a task needs to complete before the deadline. For ease of implementation, this work assumes non-preemptive tasks, although we plan to relax this assumption as part of our future work.

As discussed in Section IV, the proposed recovery mechanism are effective as long as no more than one bit error occurs in any of the memory subsystems (SRAM, SPM, Flash) every two periods of any task. It is estimated that the FIT of SRAM memories is about 0.001, i.e. on average one bit upset is observed every 10^{11} hours of operation [10]. Flash memory, on the other hand, shows low SER susceptibility [35]. Since the period of a real-time task is typically tens or hundreds of milliseconds long, we deem this assumption to be satisfied in the vast majority of embedded systems.

f) Error Free OS Data Structures: Error correction techniques to recover from faults that affect OS memory require special attention and are not within the scope of this work. For this reason, we intend to study to what degree it is possible to predictably recover faults in OS data structures as a part of our future work. A promising approach in this context is system check-pointing, i.e. periodically saving the system state of OS and copying it into a more reliable piece of memory. Further research is required to seamlessly integrate check-points into our SPM management scheme.

IV. Proposed Operating System Design

Based on the assumptions and model presented in the previous section, in this section we describe the details of the presented OS incorporating the proposed error recovery mechanisms.

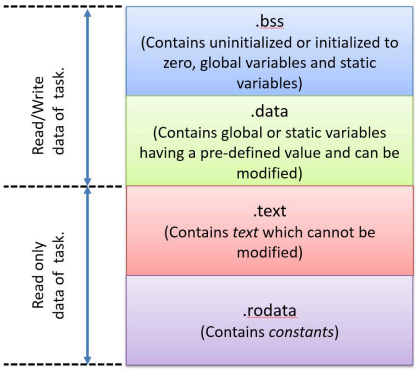

Fig. 2. Different sections of a task.

A. Background of SPM-Centric OS

We hereby provide a brief background about the previously proposed SPM-centric OS. We then introduce the new attributes required for the proposed recovery mechanisms within SPM-centric OS. Before, we delve into the details of that, let us introduce the structure of the tasks that run in our system. In particular, we describe how the tasks are structured and what are the required data that need to moved to/from the scratchpad upon task load/unload.

The task footprint contains the read-only and read/write (R/W) data sections. The read-only part of the task contains the `.text` and `.rodata` section of the task, whereas, the R/W data include the `.bss` and `.data` sections of the task. The stack of the task is created at run-time inside the scratchpad memories (SPM). At boot time, the read-only part of the task is copied from flash to main memory (SRAM). Moreover, read/write data are also allocated and appropriately initialized inside the SRAM. Figure 2 provides an overview of the described task memory layout.

In the previously proposed SPM-centric OS, the applications are never executed from the main memory but instead the following methodology is used: (1) task images are permanently stored in the flash and are loaded into the SRAM at boot-time; (2) a dedicated DMA engine is used to move task images to/from SPM upon task activation; (3) a secondary DMA engine is used to perform the I/O transfers between the devices and I/O core; (4) tasks always execute from the SPM; (5) only task-relevant I/O data are transferred upon tasks load from I/O subsystem. Doing so helps achieving predictability by ensuring conflict-free execution of the tasks from the local memories. It also allows to exploit high-speed scratchpad memories. We refer to the capability of our SPM-centric OS to dynamically move applicative tasks in and out of the SPM memories as support for *relocatable tasks*.

In our SPM-centric OS, we move tasks in and out of the I/O core using TDMA based scheduling of the DMA. Since there are M application cores and one DMA module, TDMA based scheduling of the DMA module is proposed. Furthermore, during each slot we either perform a load or unload. The resulting scheme is referred as *split-TDMA* based scheduling of the DMA module. This is depicted in Figure 4. The rules to load/unload a task during one particular slot of TDMA are as follows:

Rule 1: If a load operation can be performed, a load operation is programmed on the application DMA;

Rule 2: If a load cannot be performed and there is a previously running task to be unloaded, an unload operation is programmed on the application DMA.

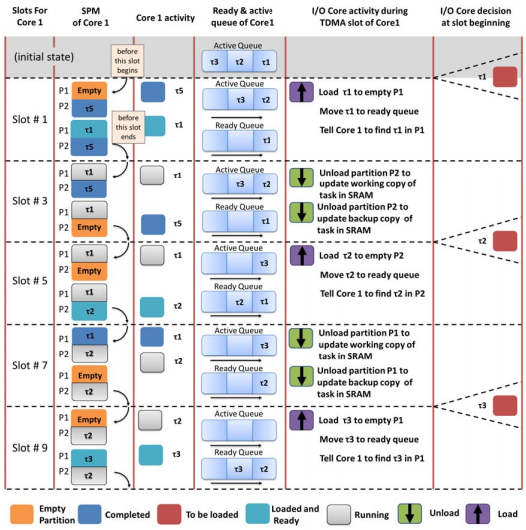

Fig. 3. Interaction between I/O Core and Core 1 for task scheduling.

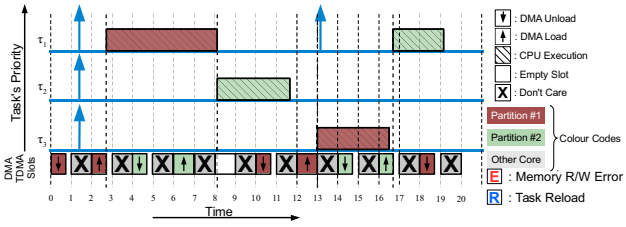

Fig. 4. Arbitrary scheduling example showing CPU, DMA and local memory

B. Integrating Error Recovery in SPM-Centric OS

The previously proposed system attains the goal of achieving predictability in a multi-core environment. However, in the event of a memory error, it does not provide any recovery countermeasure. In order to augment the predictable SPM-centric OS to recover from a memory error, we largely leverage on the existing redundancy that is built-in by design in multi-phase task system.

In fact, in our system any read-only data of a running task (in SPM) is duplicated in SRAM. On top of that, two copies of the R/W portion of the task are kept inside the SRAM. Although one can also keep two copies of the read-only data of the task inside the SRAM, this is not required because an additional copy of the read-only data is always available in flash.

The redundant copies of a task inside the SRAM are used to recover the system from a faulty state and allow to correclty handle one error every two periods of the same task in any of the memory modules. First, we describe how the overall system with redundant task copies works, then we explain how a fault in each of the memory modules (i.e. SRAM, SPM and flash) is handled.

During normal operation of the OS, both the two copies of R/W data and one read-only copy of the task in the system are the working copies. The OS data structures are initialized with proper information about both copies as working copies and one of them is marked as currently being used. When a task becomes active on an applicative core and the TDMA slot for this core arrives on the I/O core, the I/O core first checks if there is an empty partition available in one of the partition of the SPM. If an empty partition is available, the I/O core

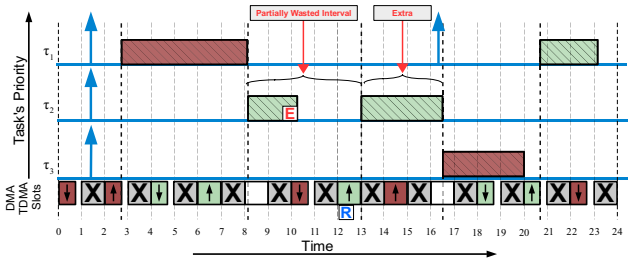

Fig. 5. The scheduling example with the proposed error recovery mechanism

programs the DMA to move the task from the SRAM copy marked as currently being utilized into the SPM. Upon DMA completion interrupt, the I/O core sends an interrupt to the applicative core for which a load is being performed. If both of the partition are found to be full and none of the task on these partition is marked as completed, then the I/O core does nothing during this slot. However, if any of the SPM partition has a task that is marked as completed. The I/O core programs the DMA to unload the task from the SPM into the SRAM. Once this operation is successful, the I/O programs another unload DMA operation to download task from SPM into the SRAM to update the second copy of the task in the SRAM. An overview of the scheduling approach in case of normal operation is depicted in Figure 3.

Unlike what depicted in Figure 4, in case of memory errors, additional operations need to be performed as shown in Figure 5. In this case, the on-chip ECC modules detect memory errors only when a read is performed on block affected by the bit flip. Upon detection of an error, the ECC modules are programmed to (i) generate an interrupt to the application cores and to (ii) report the memory address where the error occurred. In our system, we consider the occurrence of memory errors in three different kinds of memories: SRAM, SPM and flash. We now describe the proposed error recovery strategies.

1) Fault happens inside the SPM: Based on when a fault can be detected, the fault inside the SPM can be categorized into two types: the first case corresponds to a fault that happens in the read-only or read/write memory of the task during its execution; in the second case, a fault in read/write memory is detected during unload.

Whenever a fault is detected, all the applicative cores receive an interrupt from the error detection logic. Upon receiving the interrupt, all the applicative cores execute ISR 1 and check if the memory location that caused the error lies within their SPM memory range. Only the applicative core whose SPM was affected by the fault further executes the ISR 1. The affected core further checks if the error happened during the execution of the task or during the unload phase. This can be determined based on the SPM partition affected by the fault, since the OS keeps track of the state and location of each task.

In case the error is detected during the execution of the task, the applicative core marks the SPM partition from where the task was executing as empty and reschedules the task with the highest priority. This guarantees that the task is reloaded at the next TDMA slot for this particular core, thus improving the worst-case response time of the task as discussed in Section V. This case is captured in ISR 1 at lines 3-5.

In the second case, the error is detected during the unload phase of the task. As previously mentioned, during an unload operation, only read/write data are copied from SPM to SRAM twice. The error can occur during the first or the second redundant copy. In the first sub-case (A), the task had correctly

ISR 1: ISR on Application Cores From Error Detecting Logic

```
1  MEMU ISR on Applicative Cores()
2      if ErrorAddress within local core SPM Range then
3          if Error during execution then
4              Mark the SPM partition as empty
5              Reschedule task with highest priority
6          else if Error during unload then
7              if Error at first unload then
8                  Update descriptor to use second copy of R/W data for
                     next task load
9                  Mark SPM partition as empty
10                 Reschedule task with highest priority
11             else if Error at second unload then
12                 Mark second copy of R/W data as faulty
13                 Handle task as successfully completed
14             end
15         end
16     end
```

terminated its execution phase, and the error is detected before the first copy of task R/W data from SPM to SRAM is completed. In this case, the second copy in SRAM is not updated, so that valid data from the previous task execution are not overwritten with faulty data. Conversely, the first (faulty) copy in SRAM is marked as faulty, so that the backup copy will be used at next reload. Next, we mark the SPM partition from where the task was unloaded as empty and reschedule the task with highest priority, like in the previous case. This scenario is handled in ISR 1 at lines 7-10. In the second sub-case (B), the error is detected inside the SPM after the first copy was successfully unloaded, and while the redundant copy was being updated. In this case, we mark the second copy in SRAM as faulty and update the OS data structures to use the first copy in the SRAM at next load. Since the task has successfully completed, no task restart is required. This scenario is captured in ISR 1 algorithm at lines 11-14.

2) Fault happens inside the SRAM: As described earlier, there are two copies of the R/W data inside the SRAM and one copy of the read-only data inside the SRAM, whereas, a second copy of for the read-only data resides in flash. An error in SRAM can be detected only when a task is transferred from SRAM to SPM (load). Potentially, the fault could be directly reported to the I/O core by registering the corresponding interrupt. For faults in SRAM, however, we do not register and interrupt with the memory error management unit. Instead, we follow a synchronous approach: at every DMA completion interrupt, the OS checks if any error was reported by the ECC circuitry. In case of a positive outcome, the faulty address is derived.

If the faulty address lays within the memory range being loaded from SRAM, two cases are possible. In case (A) the error affected the copy of R/W data used for the transfer, the task is not marked as "ready" and its descriptor is updated to repeat another load at the next TDMA slot using the backup R/W data copy. In the second scenario (B), the fault affects the read-only data of the task in SRAM. In this case, the I/O core directly copies the word that was corrupted by the error from flash to both the SRAM and the SPM. No task re-load needs to be performed. The complete procedure handling cases A and B is described in ISR 2.

3) Fault happens inside the Flash: The flash in our system is used during the bootup process. We keep two copies of the OS image in flash. At bootup, we bring the task read-only data from flash into SRAM. Moreover, we also allocate the R/W data of each task in SRAM. At anytime during the bootup, if an error is detected in the first working copy of the flash, we switch to the second copy of the OS image in flash. Next, we

ISR 2: DMA completion Interrupt

```
1  DMA Completion Interrupt()
2      if DMA completion interrupt for load then
3          if Unrecoverable error bit is set then
4              if Error in SRAM R/W data region then
5                  Directly copy the word that caused error from backup
                     SRAM copy to faulty SRAM copy as well as to the
                     SPM
6              else if Error in SRAM read-only data region then
7                  Directly copy the word that caused error from flash to
                     SRAM as well as to the SPM
8              end
9          Resume regular operations for DMA completion – such as IPI to
             application cores after moving task into ready queue.
10     end
```

repeat the bootup procedure from the new location in flash.

V. Schedulability Analysis

Based on the description in Section IV, in this section we derive a safe bound on the worst case response time for the task under analysis τ_i. Since we follow a similar execution model as in [12], we employ the same analysis framework introduced in previous work. However, since the recovery mechanism is novel in this work, we need to account for the recovery overhead in the analysis. Therefore, in this section we first briefly summarize the response time calculation method of [12], then we detail the new cases due to recovery and prove the analysis correct.

During the analysis we assume that $\tau_i.c$ is the adjusted worst-case execution time of τ_i which includes all overheads, such as the context-switch and the regular ISR handling in the applicative-core. In this section, for simplicity, we discuss the case in which only one memory error can occur in two consecutive period of any task. However, the analysis could be easily extended to account for more frequent errors and with more than $M = 2$ cores.

Figure 6 depicts an illustrative example of the worst case scheduling scenario (critical instant at time $t = 2$ and following busy interval) for an example task set where τ_3 is the task under analysis. The schedule depicts a busy period where τ_3 suffers interference from two higher-priority tasks, τ_1 and τ_2. As in [12], we consider the busy period as composed by a sequence of *scheduling intervals* $Interval_2, Interval_3, Interval_4$ (each bounded by bold vertical lines in the figure), followed by a *final interval* $Interval_F$. During each scheduling interval, only one blocking or interfering task runs. During the final interval, the task under analysis runs. Each scheduling interval always starts with a CPU execution and ends either when the CPU finishes executing the task or when the next task finishes being loaded by the DMA, whichever happen last; at this point, the next interval starts with the execution of the loaded task. The final interval starts with the execution of the task under analysis and finishes when the task under analysis is unloaded.

We say that a scheduling interval is *CPU-bound* when it ends with CPU execution (ex: $Interval_1$, $Interval_3$ and $Interval_4$ in the figure), and *I/O-bound* when it ends with DMA load operation (ex: $Interval_2$). The length of a scheduling interval is the maximum between the execution time of the task running in the interval and the DMA operations required to load the next task. We denote the size of the TDMA slot as σ; in the worst case a load / unload operations can occupy the entire slot, including the I/O-core ISR2 DMA handling which might copy from/to the secondary copy in SRAM, Flash and the SPM. For this reason, We upper bound the length of DMA operations as a multiple of σ.

A. Response Time Calculation

Building on the above-mentioned definitions, we can follow the same technique detailed in [12] to compute the worst case response time for a task under analysis τ_i (τ_3 in Figure 6). In [12], the response time of τ_i is computed by adding three components: (1) the blocking time B caused by a lower priority task that starts executing before the beginning of the busy interval; this is $Interval_1$ executing task τ_5 in the figure; (2) the interference H comprising the remaining scheduling intervals in the busy period, which are $Interval_2, Interval_3$ and $Interval_4$ in the figure. The number of such intervals is equal to the number of interfering higher priority jobs plus one, since an extra lower priority job that starts loading before the beginning of the busy period (τ_4 in the figure) can execute within the busy period itself; (3) the worst case length F for the final interval $Interval_F$ during which the task under analysis is executed, up to the finish time for the unload operation of τ_i. Therefore, the response time of the task under analysis is $R_{\tau_i} = B + H + F$; since the length H of the interfering intervals depends on R_{τ_i}, this is computed using a standard iterative method. In particular, notice that the number of interfering higher priority jobs is computed based on $R_{\tau_i} - F$ rather than R_{τ_i}: once τ_i starts executing in $Interval_F$, newly arriving higher priority jobs cannot delay its execution anymore.

As proved in [12], the critical instant is produced when the task under analysis τ_i and all higher priority tasks arrive immediately after a lower priority task has started loading into a partition, and the other partition was loaded with another lower priority task as late as possible (i.e., two slots before). Based on the critical instant, we then obtain $B = \max(\tau_l.c, 2 \cdot \sigma) - \sigma$, where τ_l is the lower priority task with the largest execution time. Finally, based on Lemma 3 in [12], the maximum length of the final interval is $F = \max(\tau_i.c + 5 \cdot \sigma, 7 \cdot \sigma)$.

Compared to [12], our solution differs as it accounts for the possible memory error/recovery that might lead to a task reschedule. We show that we can use the same response time iteration as in [12] to calculate the response time of the task under analysis after extending H or F depending on when the memory error/recovery takes place.

B. Accounting for Error Recovery

Since we assume that no more than one error can occur for any two consecutive periods of any task, it follows that during the busy interval of the task under analysis τ_i there can be at most one task that suffers one error. A failed task is then rescheduled within bounded time.

Lemma 1: A failed task that is rescheduled with highest priority will be reloaded after at most M TDMA slots.

Proof: Since the failed task will be raised to be the highest priority task in the system and the load has priority over unload, it is guaranteed to reload the failed task in the same partition during next TDMA slot of the corresponding core, which is once every M slots. Therefore, the failed task is guaranteed to be reloaded after M TDMA slots, 2 in the case of 2 cores as shown in Figure 6. ■

At the task level, the error might occur during the load, execute, or the unload of the task. However, to produce the worst-case workload induced by a task to the schedule, the memory error must happen as late as possible during the unload of the task.

Lemma 2: A task generates the worst-case workload induced into the busy interval when the memory error occurs

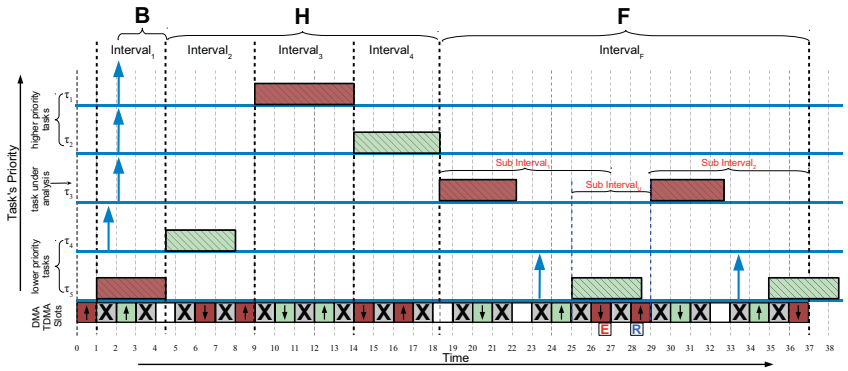

Fig. 6. SPM-centric OS task scheduling. Scheduling intervals are highlighted.

as late as possible, i.e. during the unload phase.

Proof: There are three cases in which the error can occur, (1) during the load, (2) during the execution, and (3) during the unload. Case (1) does not incur any extra overhead, since the error is recovered by ISR2 on the I/O core and the overhead is included in the TDMA slot size. However, in case (2), the execution of the task is aborted (hence partially wasted) and the task has to be rescheduled to load again after M TDMA slots. Finally, in case (3) the task is fully executed and unloaded before being rescheduled and reloaded after M TDMA slots; since this case results in the most (wasted) time added to the busy interval, it is the worst case. ■

Based on Lemma 2, we assume that a memory error always occurs as late as possible during the unload phase of a task to capture the worst case.

At the schedule level, we classify the memory error/recovery based on when it occurs with respect to the task under analysis, (1) prior to the final interval in which the task under analysis runs or (2) during the final interval.

1) Error Recovery Prior to The Final Interval ($Interval_F$):

Lemma 3: For an error that occurs prior to F, adding an extra interfering interval to H executing the task in the system with the largest execution time other than τ_i leads to the worst case response time for the task under analysis.

Proof: By definition, the error has to be in H or B to affect the response time of the task under analysis. Based on Lemma 2, the error should occur during the unload of a task; hence, the recovery mechanism forces the failed task to be rescheduled, i.e., a new scheduling interval is added to the schedule. The rescheduling of the failed task will account for the execution and the load/unload operations of the failed task. Furthermore, regardless of the tasks priority, the rescheduled (failed) task always runs as the highest priority in the system, thus this additional scheduling interval causes interference to the task under analysis.

Since we cannot make any assumption on which task might fail, lower-priority task or higher-priority task, it is safe to assume that the longest executing task in the system other than the task under analysis will be scheduled to run in the induced interval. Finally, note that even if the task that fails is the one executed in B, the rescheduled task will execute during H including its load/unload memory operations. Since the algorithm in [12] is able to correctly upper bound the interference in H caused by any task without making any assumption regarding their order, we then simply add the induced interval to H to capture the worst-case response time for τ_i. ■

Based on Lemma 3, let H_{rec} be the computed length of interfering intervals including one restarted task. To give a concrete example on how an error happening prior to the final interval ($Interval_F$) extends H, refer to Figure 7. In this example, τ_5 fails during unload; it is reloaded at time 8-9 and executes in a new third interval in place of τ_1; τ_1 then executes in the fourth interval and τ_2 in a fifth interval. Since our analysis bounds the length of the intervals in H without making any assumption on the order of the tasks, this is safe. Note that the analysis is based on H always having tasks ready to load by definition, which is not the case for $Interval_F$, thus requiring a separate analysis in Lemma 4.

2) Error Recovery in The Final Interval ($Interval_F$):

Lemma 4: For an error that occurs within $Interval_F$, the maximum length of the interval with $M = 2$ is $F_{rec} = 2 \cdot \max(\tau_i.c + 5 \cdot \sigma, 7 \cdot \sigma) + \max(\tau_u.c, 4 \cdot \sigma) - 2 \cdot \sigma$, where τ_u is the task in the system with the largest execution time other than τ_i.

Proof: As defined earlier, $Interval_F$ starts with the execution of the task under analysis τ_i and finishes with the end of the unload phase of τ_i. Based on Lemma 2, in the worst case, the error occurs during the unload phase. Therefore, the error must occurs in the unload phase of τ_i, otherwise τ_i will unload successfully and the interval finishes.

Based on Lemma 3 in [12], in the normal case, the unload operation for τ_i completes after at most $max(\tau_i.c+5\cdot\sigma, 7\cdot\sigma)$ time units from the beginning of the interval; in Figure 6, this occurs at time 27. However, in the case of memory error during the unload operation and task recovery, $Interval_F$ is extended. To simplify the computation of the worst-case length F_{rec} of $Interval_F$ accounting for recovery, we divide the interval in three sub intervals. The first execution of τ_i is contained in the first sub interval $SubInterval_1$, which finishes with the first (failed) unload of τ_i. The last sub interval $SubInterval_2$ starts with the second execution of τ_i after the reload and ends with the (successful) second unload, time 37 Figure 6. In the worst case, there can be a middle interval $SubInterval_u$ in which another task τ_u is loaded into the other partition and executed.

Based on Lemma 1, τ_i will be reloaded after M TDMA slots (2 in our case). $SubInterval_u$ finishes and $SubInterval_2$ starts either when the reload operation is complete (case shown in the figure) or when τ_u finishes, whichever happens last. Since in the worst case $SubInterval_u$ starts after τ_u has been loaded, and we need M TDMA slots for the failed unload and M TDMA slots for the reload of τ_i, the length

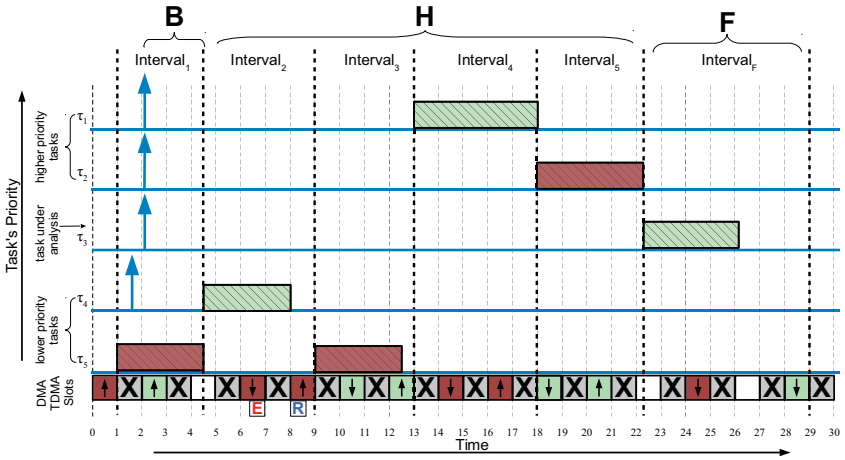

Fig. 7. Example showing how H is extended by an induced interval due to error recovery prior to $Interval_F$

of $SubInterval_u$ can be upper bounded as $\max(\tau_u.c, 4 \cdot \sigma)$ when $M = 2$. The lengths of the sub intervals $SubInterval_1$ and $SubInterval_2$ are calculated the same way as in Lemma 3 in [12] as discussed above. However, as shown in Figure 6, $SubInterval_1$ and $SubInterval_U$ overlap by two TDMA slots (time 25 to 27 in the figure). This is because the length of $SubInterval_1$, as in Lemma 3 in [12], is calculated up to the end of the unload phase, while $SubInterval_U$ starts $M = 2$ slots before. To overcome this overlap, we subtract the overlapped time which is $2 \cdot \sigma$ when $M = 2$.

As a result, the length of $Interval_F$ is computed as:

$$\begin{aligned} F_{rec} &= SubInterval_1 + SubInterval_U \\ &\quad + SubInterval_2 - 2 \cdot \sigma \\ &= \max(\tau_i.c + 5 \cdot \sigma, 7 \cdot \sigma) + \max(\tau_u.c, 4 \cdot \sigma) \\ &\quad + \max(\tau_i.c + 5 \cdot \sigma, 7 \cdot \sigma) - 2 \cdot \sigma, \end{aligned}$$

which is the same as the value in the hypothesis. ■

Finally, we have to account for the overhead of executing the recovering ISR 1 on the applicative core during the busy interval. If we let ρ be the maximum length for the ISR, we can compute a safe upper bound by simply adding ρ to the response time iteration. In general, we do not know which case will lead to the worst response time calculation for τ_i, when the error occurs before $Interval_F$ (case 1) or within $Interval_F$ (case 2). As a result, we independently calculate both iterations:

$$R^1_{\tau_i} = \rho + B + H_{rec} + F, \tag{1}$$

$$R^2_{\tau_i} = \rho + B + H + F_{rec}, \tag{2}$$

and take the maximum response time among $R^1_{\tau_i}, R^2_{\tau_i}$. In particular, note that it is easy to see that $F_{rec} - F$ is larger than the size of the additional interval added for case 2. This is the main reason why we chose to rescheduled failed tasks at the highest priority: it minimizes the worst case length of $Interval_F$ in case the task under analysis fails. On the other hand, rescheduling the task at higher priority means that we have to consider any task, rather than just higher priority tasks, for the extra interval in case 2, but as discussed this is generally not the worst case. However, note that we cannot formally avoid computing the iteration in Equation 1 because the interfering window for higher priority jobs is based on $R^1_{\tau_i} - F = \rho + B + H_{rec}$, which is larger than $R^2_{\tau_i} - F_{rec} = \rho + B + H$.

VI. Implementation

In this section, we will describe the implementation of the recovery mechanism for SPM-centric OS using component-off-the-shelf (COTS) MPC5777M embedded platform.

A. Architectural Overview of Considered Platform

First, we summarize the architectural features of the considered MPC5777M processor that is compliant with the hardware assumptions made in Section III. A brief summary of these features is provided in Table II.

TABLE II
Characteristics of Freescale MPC5777M SoC

Chip Name	MPC5777M (Matterhorn)
Manufacturer	Freescale
Architecture	Power-PC, 32-bit
CPU Unit	2x E200-Z710 + 1x E200-Z709 + 1x E200-Z425 (I/O)
CPU Frequency	Application Cores (300 Mhz) I/O Core (200 Mhz)
Processing Unit	CPUs, DMA, Interrupt Controller, NIC
Operational Modes	Parallel + Lockstep (on one applicative core)
ECC Protection	Cache, RAM, Flash Storage
Cache Hierarchy	L1 (Private Instructions + Data) + Local Memory
Local Memory (SPMs)	Instructions (16 KB) + Data (64 KB)
L1 Cache Size	Instructions (16 KB) + Data (4 KB)
SRAM Size	404 KB
Flash Size	8 MB
Main Peripherals	Ethernet, FlexRay, CAN, I2C, SIUL
MEMU	MEMU For SRAM, Peripheral RAM and Flash

In the considered platform, we have scratchpad memories for each core. There is also an error correcting codes (ECC) logic implemented in the hardware that provides single bit error correction and double bit error detection (SEC-DED). The module implements Hsaio Codes for detection and correction, this means that both correction and detection can be simultaneously done. A DMA module to move data between different memories is also present. A separate I/O core is provided to handle the traffic from I/O peripherals. In order to test and verify the safety features of the SoC, the chip also implements fake error injection mechanisms that are helpful to verify the reaction to various faults. We use these fault injection mechanisms to evaluate our system.

B. OS-level Integration of Recovery Mechanisms

The proposed error recovery mechanisms for SPM-centric OS were implemented using Evidence Erika Enterprise[1]. Erika Enterprise is an open-source RTOS that is compliant with the AUTOSAR[2] (Automotive Open System Architecture) standard. AUTOSAR is an open standard for automotive architectures providing a basic infrastructure for vehicular software. Erika Enterprise features a small memory footprint, supports multi-core platforms and implements common scheduling policies for periodic tasks. We performed a porting of Erika Enterprise on the MPC5777M MCU, adding support for UART communication interface, interrupt controller, caches, memory protection unit (MPU), data engines (DMA), Memory error management unit (MEMU), FCCU and Ethernet controller.

In order to implement our SPM-centric OS, we have augmented Erika Enterprise to support position-independent (relocatable) tasks. We rely on the compiler[3] support for `far-data` and `far-code` addressing modes. In this way, tasks are compiled to perform program-counter-relative jumps and indirect data addressing with respect to an OS-managed base register. We have extended the default task loader to exploit DMAs for transferring task images from SRAM to local memories and vice-versa. Similarly, the OS scheduler has been adapted to implement the strategy discussed in Section IV.

In Erika Enterprise, tasks are compiled and linked directly inside the image of the OS. For each task in the system, Erika-specific meta-data need to be defined. Additionally, meta-data that extend the task descriptors for SPM-centric operations are required. Manually configuring these parameters is tedious and error-prone; hence, we developed an OS configurator. The tool uses high-level task definitions and generates the final configuration for our SPM-centric OS. Specifically, each core is associated with a set of configuration files that describe: number of tasks, their priority, task entry points, initial status and so on. When a task is added, these files need to be configured accordingly.

First, the body of all the tasks is placed in an ad-hoc file. Similarly, task-specific data that need to be preserved across activations are defined in different files and surrounded with appropriate compiler-specific `PRAGMA`. This is fundamental to ensure that: (A) specific linker section is used to store task code and data images; and (B) position-independent data and instructions are generated. A separate file also defines the relocatable task table, which stores the status of each relocatable task. This structure includes: (A) position in SRAM of the task code and data images; (B) current status of the task (e.g. loaded, completed, unloaded); (C) SPM partition of last relocation.

Finally, appropriate sections to place data and code for tasks need to be added in the linker script for the I/O core only. The developed tool is able to perform this operation automatically upon modification of any of the user-defined tasks. The DMA is responsible for copying the required task image, on behalf of any of the applicative cores, to the scratchpad memory of the requesting core.

In the considered MPC5777M platform, there is a memory error management unit (MEMU) that is responsible for collecting and detecting the faults in different memory subsystems such as SRAM, SPM and Flash. The MEMU implements separate tables for reporting correctable and uncorrectable errors. There are separate tables for each kind of memories. These tables contain the address of the fault that caused error, moreover, there is a register inside the MEMU that tells if the fault that occurred is a correctable or uncorrectable fault. On MPC5777M, there is no way for the MEMU to send an interrupt to the CPU in case of a fault. There is a separate FCCU module present on the chip that collects all the errors that are forwarded to it from the MEMU. The FCCU module can be preprogrammed to take certain actions based on a particular error. Moreover, it is also responsible of generating interrupt to the processor to notify it in case any kind of errors that are being reported to it from the MEMU. Figure 8 shows how different modules are connected to each other.

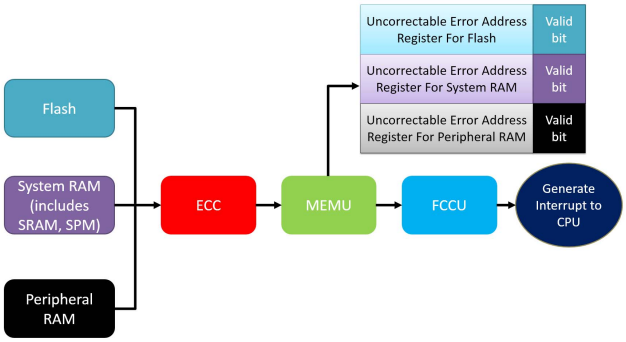

Fig. 8. Block diagram of error handling circuitry.

In order to detect the faults in the SPM, we registered a FCCU interrupt with application cores. This interrupt gets generated when an error is reported by the MEMU to the FCCU in one of the memory subsystem. Upon interrupt generation all the application cores receive an interrupt check if the address that caused the error belongs to the its own SPM address range. If not the application core exists the ISR and continues whatever it was doing. In case the address matches SPM range it checks whether the fault is in the read-only or read-write data of the task. In case of read-only errors the core copies the word that caused error and starts re-executing again from the point where the error occurred. In the later case, the application core marks the partition as empty and reschedules the task. The procedure is exactly same as we proposed in the OS design section. This procedure only applies for errors in the SPM. The errors for SRAM and Flash are handled by the I/O core and they follow same procedure as the on presented in the OS design section.

VII. Evaluation

In order to validate the results of the proposed error recovery mechanisms, we performed a variety of experiments. We measured the overheads of different recovery handling routines to account for the OS overhead. We then present the results of the EEMBC benchmarks by running the tasks from the SRAM versus SPM. Based on the collected data we then calculate schedulability graphs.

A. SPM-Centric OS Overhead Evaluation

The most important parameter of our proposed system is the size of the TDMA slot. The slot needs to be long enough

[1] http://erika.tuxfamily.org/drupal/

[2] http://www.autosar.org/

[3] Applications and OS are compiled using the WindRiver Diab Compiler version 5.9.4 - http://www.windriver.com/products/development-tools/

TABLE III
DETAILS OF OS PARAMETERS

Parameter	Time (μs)
DMA Load time (Largest Code, R and R/W Data size)	209
DMA Unload time (Largest R/W Data size)	61.5
DMA setup	3.16
Context switch	0.46
Minimum ISR overhead on Applicative cores	0.70
Maximum ISR overhead on Applicative cores	8
Maximum ISR overhead on I/O core for DMA completion	2
TDMA slot size	215

to account for the load/unload of any task in the system. In order to calculate the upper bound, we restrict the slot size to be greater or equal to the footprint of the task with maximum size among the benchmarks. In addition, based on our recovery mechanism, the TDMA slot size must be long enough to allow two unloads of the completed task. Therefore the TDMA slot size is equal to $\max(worst\_task\_load, 2 \cdot worst\_task\_unload) + IO\_core\_ISR\_overhead$. Table III shows the system parameters including DMA times and TDMA slot size.

We have measured the DMA modules configuring overhead. In addition, since the interrupt from the memory error management unit is delivered to all the application cores, we have measured the overhead of ISR1. This ISR has minimum and maximum overhead. The minimum overhead occurs when the ISR only checks if the error address is relevant or not and concludes by its irrelevance. On the other hand, the maximum overhead is experienced in the case of a relevant memory fault. We also measured the maximum overhead of I/O core ISR2 for DMA completion interrupt. All of these results are reported in Table III.

B. Results of EEMBC Benchmarks

In order to evaluate our system, we run the EEMBC benchmark (automotive suite) on the platform. We compared the execution time of running the reported applications in Table IV out of the local SPM and the main SRAM memory. When the applications run from the main memory they suffer contention delay due to the shared access to the main memory, refer to Table IV for more details. Here we have updated the benchmark table to explicitly mark the sizes of the read-only data and read-write data of the applications as it is relevant to our recovery mechanism. The results are shown in Table IV.

TABLE IV
DETAILS OF EEMBC BENCHMARKS.

Benchmark	SPM Time (μs)	SRAM Time (μs)	Relocatable Code Size (bytes)	Read only Data Size (bytes)	Read/Write Data Size (bytes)
tblook	1013	1015	1892	10916	60
matrix	1053	1054	4774	12188	124
a2time	1002	1029	2538	1704	148
pntrch	1036	1145	1398	4800	128
ttsprk	383	425	4772	2592	4848
iirflt	1040	1189	3512	888	248
canrdr	1009	1359	1562	12276	56
bitmnp	990	1389	3282	72	1494
rspeed	1012	1457	1208	13200	40
puwm	1036	1540	2500	2400	180
aifirf	1005	1564	2286	1120	84
aifftr	916	1642	4458	2304	1912
aiifft	1170	2092	3540	3072	1656
idct	1045	2126	4690	244	1788
Total size			42412	67776	27766

TABLE V
SPACE OVERHEAD OF HW VERSUS SW RECOVERY

	HW SEC-DED (bits)	HW DEC (bits)	SW DEC (bits)
Number of extra bits required for R/W data (12,766 bytes)	12,766	22,344	127,660
Number of extra bits required for R-only data (42,412 + 67,776 bytes)	110,192	192,836	110,192
Total number of extra bits	80,542	215,180	237,852

C. Overhead of Software versus Hardware Recovery

The proposed software-based recovery technique is able to correct errors that can be detected (only) but not corrected by the hardware. If the hardware provides double-bit error correction (DEC) capabilities, then our approach is redundant and should not be used. However, if the hardware, as it is often the case, supports only SEC-DED, then we introduce a timing penalty to recover 2-bit errors that are detected but not corrected by the hardware. Under SEC-DED, 1-bit errors are still corrected only by the hardware. Introducing a timing overhead effectively means trading schedulability for reliability. We quantify this trade-off in Figure 9 and 10. Clearly, when software-based recovery is used, the time overhead can be significant. Nonetheless, we believe that it is reasonable to trade a portion of the CPU utilization to make sure that safety-critical tasks produce correct outputs.

Let us focus on the typical ECC support provided by commercial hardware, i.e. SEC-DED. We hereby compare the **overhead in space** introduced by our software-based recovery mechanism with an equivalent hardware-based implementation, i.e. DEC support in hardware. Introducing DEC support in hardware is possible, but its implementation is complex and costly. In fact, it has been shown [36] that the ASIC area can grow up to 13x. The software approach, conversely, can be deployed on less expensive hardware and only requires to keep redundant copies of the R/W portion of application memory. The SEC-DED requires 8 check bits to implement the detection and correction mechanism for a 64-bit data word, while DEC requires 14 check bits to correct a double bit error [36] in a 64-bit data word. Using this information, we compute the number of extra bits required for hardware-based (HW) SEC-DED, hardware-based DEC, and our proposed software-based (SW) recovery mechanism for DEC in Table V. In the table, we assume that our taskset is comprised of all the benchmarks in Table IV. First, note that our technique relies on SEC-DED and only duplicates R/W data and ECC bits. For read-only (R-only) memory, only the extra bits required to implement SEC-DED are considered, because a second copy of read-only data is always available in less expensive flash memory. Moreover, for a fair comparison, we assume that HW DEC support is only provided for application memory, instead of the whole main memory. In this setting, the SW overhead is about 10%. We argue however that (i) not requiring additional ECC check logic in the SW approach partially offsets the cost for the additional memory bits; and that (ii) if extra DEC bits were considered on the whole main memory, the SW approach is to be preferred in terms of overhead: in fact, with 512 KB of main memory, DEC would require about 917,504 extra bits.

Next, we consider the **overhead in time** for hardware-based recovery and the proposed software-based technique.

According to [36], hardware implementations of SEC-DED introduce a latency of 1.3 ns, whereas DEC implementations introduce a latency of 2.2 ns on every memory transaction. Our SW approach on the other hand still requires SEC-DED, but only introduces a time overhead if a double-bit error occurs. Depending upon which part of the task is affected by the two-bit error, the recovery overhead differs. For instance, an error in the read-only data only requires one word to be copied from a redundant copy, whereas, an error in read/write data of the task in the worst-case may require the entire task to be re-loaded and re-executed. It is hard to compare the time overhead of software-based and hardware-based approaches, because it depends on the number of memory accesses performed by the tasks under analysis. Nonetheless, we expect that hardware-based DEC implementation can be more efficient in time. As vendors typically only provide SEC-DED support, however, we believe that offering the choice at design time to recover double-bit errors in software still represents a valuable contribution in the context of safety-critical systems. We discuss how software-based recovery impacts system schedulability in Section VII-D.

D. Results of Schedulability Analysis

For the schedulability evaluation of our newly proposed scheme we first present the schedulability curve of normal case when there are no errors and compare it with the case when we have errors. We also show the contention based approach where we have no error recovery. The case referred as "contention" corresponds to the case where no scratchpad management is implemented and in which tasks execute directly from SRAM contending for bus access.

We computed the response time of the same workload for the three cases: the traditional SPM-centric OS mechanism with no errors as proposed in [12]; the augmented SPM-centric OS with error recovery mechanisms using the analysis in Section V and with artificial error injection; and the contention based execution using standard response time analysis. For the evaluation, we have considered the EEMBC benchmarks in Table IV and the overheads in Table III.

For a given system utilization, each application is randomly selected and assigned a random period in the range between 10 ms to 100 ms. The tasks utilization is then computed based on the measured execution time of the applications and the selected period. Tasks are randomly generated until the sum of the individual tasks utilizations reaches the required system utilization.

Figure 9 shows the result of the schedulability analysis of our newly proposed SPM-centric scheme with error recovery and compares it with the case when there are no errors. Based on the figure, we can conclude that there is limited degradation in schedulability for supporting the recovery mechanism. This degradation can be justified by the fact that the system is both predictable as well as fault tolerant. Moreover, from the Figure 9 we can also see that our SPM-centric approach with error recovery still performs significantly better than the contention-based case where no error recovery is performed.

Figure 10 shows the system utilization when 50% of the task sets are schedulable. The X-axis represents the window of periods used to generate the task sets. As expected in any non-preemptive scheduler, with tight periods all three mechanisms degraded due to the blocking time. However, with error recovery the degradation is more severe in the case of very small periods due to the extra overhead paid

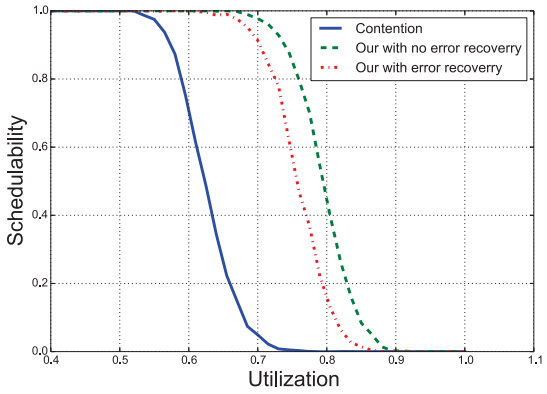

Fig. 9. Schedulability degradation when applying the recovery mechanism

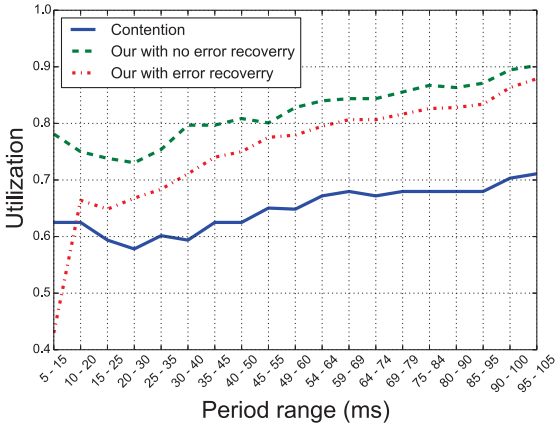

Fig. 10. Utilization degradation as a function of tasks periods

for recovery. In most cases, with error recovery, the proposed strategy achieves better utilization compared to the contention-based approach. Additionally, the loss in utilization arising from additional error recovery overhead remains within an acceptable range, and marginally decreases for larger task periods.

VIII. Conclusion and Future Work

In this paper, we have presented error recovery mechanisms to recover from bit flips that cannot be recovered by hardware modules by introducing redundant copies of the tasks in the system. We integrate these strategies into the SPM-centric OS, which by design supports redundancy to achieve predictability. We exploit this design feature of the SPM-centric OS by adding some more redundancy to make it fault tolerant to bit flips errors. The result is a predictable SPM-centric OS which is also fault tolerant to bit flips.

In our current work, we have considered the case in which only one memory error can occur in two consecutive period of any task. However, the analysis could be extended to account for more frequent errors. Similarly, we discuss the case with $M = 2$ cores, since it is used in our prototype, however, the analysis could be extended to account for a larger number of cores. As a part of our future work, we intend to investigate the aforementioned extensions from both an analytic and implementation standpoint.

ACKNOWLEDGMENT

The material presented in this paper is based upon work supported by Hitachi America Ltd. under contract Hitachi 2013-07132, the National Science Foundation (NSF) under grant numbers CNS-1302563 and CNS-1646383, NSERC DG 402369-2011 and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF and other sponsors.

REFERENCES

[1] FAA position paper on multi–core processors, CAST32 (rev 0). http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast32.pdf. Accessed: 2015-01-26.

[2] R. Mancuso, R. Pellizzoni, M. Caccamo, Lui Sha, and Heechul Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183, July 2015.

[3] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. Real-time computing on multicore processors. *Computer*, 49(9):69–77, Sept 2016.

[4] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-critcality provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[5] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Architecture of Computing Systems-ARCS 2011*, pages 122–134. Springer, 2011.

[6] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013.

[7] J. Whitham and N.C Audsley. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 3–12. IEEE, 2012.

[8] L. Bathen and N. Dutt. Exploiting unreliable embedded memories. In *International Symposium on Electronic System Design*, December 2011.

[9] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 411–420, New York, NY, USA, 2006. ACM.

[10] C. Slayman. Whitepaper on soft errors in modern memory technology. Technical report, Ops A La Carte LLC., Santa Clara, CA, 2010. URL http://www.opsalacarte.com/pdfs/Tech_Papers/DRAM_Soft_Errors_White_Paper.pdf.

[11] D. Sperling and D. Gordon. *Two Billion Cars: Driving Toward Sustainability*. Oxford University Press, 2009.

[12] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

[13] D. Bui, E.A. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference (DAC)*, pages 274 – 279, June 2011.

[14] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. ISSN 0272-1732.

[15] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), November 2015.

[16] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126, Dec 2006.

[17] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. Rtos support for multicore mixed-criticality systems. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.

[18] G. Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computers*, 52(10):1332–1346, Oct 2003.

[19] S. Punnekkat, A. Burns, and R. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, 2001.

[20] M. A. Haque, H. Aydin, and D. Zhu. Real-time scheduling under fault bursts with multiple recovery strategy. In *19th Real-Time and Embedded Technology and Applications Symposium*, pages 63–74, April 2014.

[21] R. M. Pathan. Fault-tolerant real-time scheduling algorithm for tolerating multiple transient faults. In *2006 International Conference on Electrical and Computer Engineering*, pages 577–580, Dec 2006.

[22] L. Jun, Y. Fumin, and L. Yansheng. A feasible schedulability analysis for fault-tolerant hard real-time systems. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 176–183, June 2005.

[23] B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *37th IEEE Real-Time Systems Symposium (RTSS 2016)*, December 2016.

[24] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu. Global edf scheduling for parallel real-time tasks. *Real-Time Systems*, 51(4):395–439, July 2015. ISSN 0922-6443.

[25] D. Compagnin, E. Mezzetti, and T. Vardanega. Putting run into practice: Implementation and evaluation. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 75–84, July 2014.

[26] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, and F.D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE International Real-Time Systems Symposium (RTSS'16)*, December 2016.

[27] S. Wasly and R. Pellizzoni. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 75–86. IEEE, 2014.

[28] Durrieu, G. and Faugere, M. and Girbal, S. and Pérez, D. G. and Pagetti, C. and Puffitsch, W. Predictable flight management system implementation on a multicore processor. *ERTSS'14*, 2014.

[29] C. Maia, L. M. Nogueira, L. M. Pinho, and D. G. Prez. A closer look into the AER model. In *2016 IEEE International Conference on Emerging Technology and Factory Automation, (ETFA 2016)*, September 2016.

[30] A. Bovenzi, J. Alonso, H. Yamada, S. Russo, and K. S. Trivedi. Towards fast os rejuvenation: An experimental evaluation of fast os reboot techniques. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 61–70, Nov 2013.

[31] F. Abdi, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *2016 IEEE International Conference on Emerging Technology and Factory Automation, (ETFA 2016)*, September 2016.

[32] C. H. A. Costa, Y. Park, B. S. Rosenburg, C. Y. Cher, and K. D. Ryu. A system software approach to proactive memory-error avoidance. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 707–718, Nov 2014.

[33] Y. Y. Chen, Y. L. Kuo, and K. L. Leu. An autonomous recovery software module for protecting embedded os and application software. In *Global High Tech Congress on Electronics, IEEE*, pages 165–169, Nov 2012.

[34] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of R and D*, 14(4):395–401, 1970.

[35] M. She. *Semiconductor Flash Memory Scaling*. University of California, Berkeley, 2003.

[36] R. Naseer and J. Draper. Parallel double error correcting code design to mitigate multi-bit upsets in srams. In *Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European*, pages 222–225. IEEE, 2008.