Implementing a Fault Tolerant Real-Time Operating System

EEL 6686: Presentation 2

Chris Morales Kaz Onishi

ECE
University of Florida, Gainesville, Florida

February 19, 2015

Introduction

- What is a real-time operating system (RTOS)?
  - OS that guarantees a certain functionality within specified time constraints
  - Link between software and embedded system
- Main roles:
  - Task management - scheduling and priorities
  - Time management - timing constraints, delays, time outs
  - Dynamic memory allocation - file creations, protections
  - Interprocess communication and synchronization - keeps data intact
- RTOS need valid results both in correctness and in specified amount of time
  - Incorrect results would result in a failure
  - Not meeting timing constraints also results in a failure

Specifications

- Should be able to react to external stimuli in timely fashion
- Emphasize predictability efficiency
- Soft deadline
  - Results in degraded performance if deadline is missed
  - Result still has utility after deadline
  - Example: Video streaming, anything with human interactions
- Firm deadline
  - Result has no utility after deadline
  - Hard deadline if missing deadline causes a catastrophic event
  - Example: Automation, medical systems

Fault Tolerant RTOS

- Some form Fault tolerance is necessary in everyday systems
- Problem: Fault tolerance usually comes with overhead
  - Design a very fault tolerant system?
    - Less failures in general but for RTOS does it really?
    - Introduces more timing constraints
    - For RTOS if deadline is not met considered a failure
  - No fault tolerance?
    - Can meet timing constraints much easier
    - If there is a fault it can also lead to system failure
    - Faults can cause bad results which cause more timing issues
- Need good balance

Fault Mitigation Techniques

- Traditional fault tolerant systems include redundancy
- Some common ones seen include:
  - Processor
    - Triple modular redundancy (TMR)
    - If one input is wrong, it is masked by voting system
  - Fault tolerance in algorithms
    - Error correction codes for data
  - Timing redundancy for transient errors
  - Memory
    - Dynamic storage allocation
    - Redundancy of disk or error correction codes
- Other considerations must be made for operating systems
- What are some fault tolerant techniques for RTOS?

Mitigation Techniques for OS (1/2)

Some examples of techniques to make OS more fault tolerant:

- Kernel Considerations
  - Should be able to communicate to supervisor to correct errors
  - Event logging to determine where error happened
  - Protection against bad system calls
- Interrupt handling
  - Should be predictable for fault tolerance
  - All services should be able to save state and execute to prioritize higher interrupts
- Memory management units
  - Some RTOS disable for higher speed
  - Everything is run on same address space not recommended
- I/O management
  - Active spares in order to not waste time

Mitigation Techniques for OS (2/2)

- Many different ways to make an OS fault tolerant
  - Cannot implement all techniques due to size/timing constraints
  - Implementations increase timing, increases chance of failure
- What to make redundant?
  - Options are limited for hard deadlines
  - Need to pick out critical functions of RTOS
  - Make only critical functions fault tolerant
  - What method to use for least timing overhead?

Critical Functionality of RTOS

Scheduling and resource allocation considered important for RTOS

- Manage resources efficiently to stay within timing constraints
- Scheduling algorithms
  - Rate monotonic - static scheduling defined in advance
  - Earliest deadline first - dynamic scheduling
  - Least laxity first - based on slack (amount of time left)
- N-copy scheduler
  - Copied n times and run simultaneously
  - Results are voted for at the end
  - Least amount of timing overhead
- Checkpointing
  - Go back to a previous known state
  - bad timing overhead
  - better hardware overhead
- Making scheduler and resource allocation fault tolerant can make sure timing constraints are met

Introduction and Implementation of a fault tolerant resource manager using ARINC 653 and RTEMS

- Process and Tread Management
  - RTOS should activate a process once and release it once
  - Each activate and release must start on time and finish before its deadline
  - RTOS must guarantee the availability of resources for required process

Possible fault due to careless management of resources

- If tasks behavior are not monitored they may execute carelessly
- Careless or malicious executions could eat up precious resources
- This could exhaust system resources

Example

- A fault in an application could create too many tasks
- Other tasks fail because of their inability to acquire resources
- Fault tolerant RTOS resource manager must exist to prevent such scenarios

Solution 1:

- Determine a process's maximum allowable resource usage before execution
- Processes are not allowed to use more than their reserved resources
- Processes wanting more than allowed are discarded as error

Solution 2: Fixed -priority systems ARINC 653

- Process manager runs partitions or address spaces, according to a timeline provided by the designer
- Each address space is placed into one or more windows of execution in a hyper period
- Tasks within an address space are selected and executed, while others a rejected
- Hard/Critical Processes are guaranteed their required resources
- Soft/normal processes are rejected

ARINC 653 Architecture 1

- Application software layer separates applications into partitions
- Services are routed through the Application Executive Interface
- Containment of faults from each partition must be ensured by the core software layer

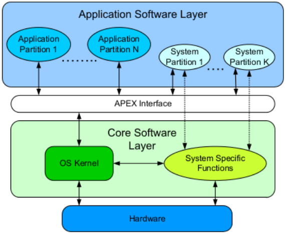


Figure 1. Standard ARINC 653 System Architecture

ARINC 653 terminology

- Spatial Partitioning: ensures one application or partition does not access anothers memory space
- Temporal Partitioning: ensures that the activities of one partition do not affect the timing of another

ARINC 653 Implementation using RTEMS

- AIR innovation initiative sponsored by ESA creators of the ARINC 653 architecture based on RTEMS
- Current available ARINC 653 implementations are commercial and very expensive
- Example: X-47B unmanned aerial vehicle owner US Air Force

RTEMS Kernel Architecture 1

- Designed to support application with real time requirements while maintaining a compatible interface with open standards
  - Multitasking capabilities
  - Event-driven, priority-based, preemptive scheduling
  - comprehensive mechanism for inter-task communication and synchronization
  - high degree of user configurability
  - supports homogeneous and heterogeneous multiprocessor system architectures

RTEMS Kernel Architecture 2

- RTEMS does not provide all the required mechanisms for ARINC 653
- However, RTEMS does meet the basis for a deployment of ARINC 653 services
- RTEMS resource manager form the executive interface presented to the application

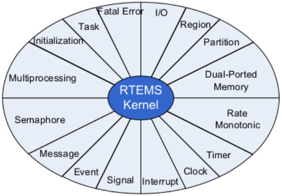


Figure 2. The RTEMS resource managers

RTEMS Kernel Architecture 3

- From the point of view of its internal architecture RTEMS provides a set of layered components that provide a set of services to real-time applications
- RTEMS is a robust multitasking operating system kernel
- RTEMS can support a wide range of processors through an adaptor layer that is independent of hardware known as a board support package

AIR System Architecture 1

- The AIR architecture preserves the hardware and and RTOS independence
- AIR adds modules to the RTOS kernel to include spatial and temporal partitioning which are:
  - AIR partition scheduler
    - Determines who owns the resource at a given time, ensures temporal segregation
  - AIR partition dispatcher
    - Saves and restores execution content for the heir partitions and guarantees spatial segregation
  - AIR inter-partition communication module
    - Allow coms between different partitions without violating spatial segregation

AIR System Architecture 2

- ARINC 653 application executive interface (APEX) furnishes the following set of services using the AIR architecture
  - partition management
  - process management
  - time management services
  - inter-partition communication services
  - intra-partition communication services
  - Health monitoring

AIR System Architecture Overview

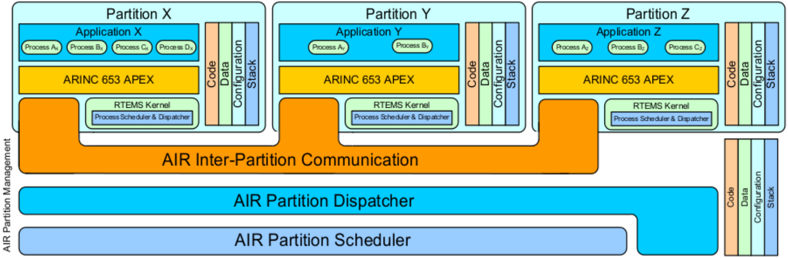


Figure 3. Overview of the AIR System Architecture

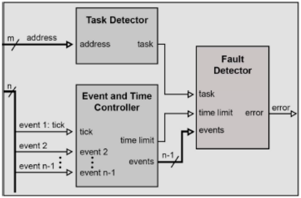
Hardware Scheduler (Hw-S)(1/2)

- Focuses on task management of an RTOS
- Hardware implementation to detect faults affecting apps
- Aims to detect faults causing:
  - Sequence errors - scheduling failures
  - Timing errors
- Assumptions for hardware scheduler:
  - Scheduler is required part that defines when to execute task
  - Algorithm is deterministic
  - Tasks implemented by programs stored in specific memory location
  - Tasks behavior follows set of time constraints and defined by external events

Hardware Scheduler (Hw-S)(2/2)

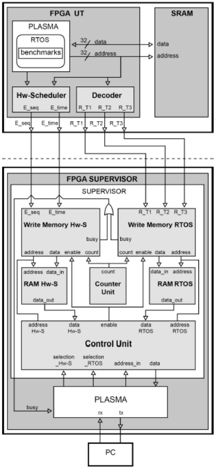
Split into 3 blocks

- Task detector
  - Based on info stored in addr table
  - Generated during compile time
  - Identifies tasks in execution
  - Reads address accessed by microp
  - Compares with records in addr table
- Event and time controller
  - Defines time limit deadline
  - Detects events that change exe. time
- Fault detector
  - Implements the scheduling algorithm
  - Fault detection based on task in exe.



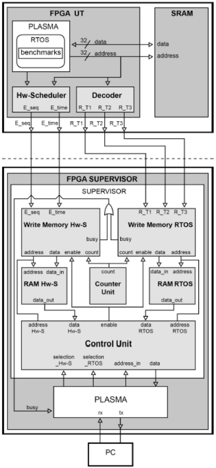
Validation (1/2)

- Testing Hw-S developed using FPGA
  - Comparing detection with RTOS vs Hw-S
  - 2 FPGAs used to gather results
  - Identifies tasks in execution
- Using Xilinx Spartan Model XC3S500E
- split into 2 parts
- FPGA under test (UT)
  - Inject faults into this device
- FPGA supervisor
  - Stores results
  - sets up fault injection campaigns



Validation (2/2)

- FPGA UT is Composed of:
  - Hw-S unit that was developed in this paper
  - 32 bit RISC plasma processor running app.
  - Unit to decode address associated to task
- FPGA supervisor composed of:
  - Control unit which receives control signals
  - RAM unit to store execution flows from Hw-S and RTOS
  - Write memory units for storing errors
  - Counter unit to indicate when data is stored



Fault Injection Setup

- Two benchmarks were tested with 3 tasks each
  - Task 1 (T1), Task 2 (T2), and Task 3 (T3) access and update 3 different global values
  - T1 sends value to queue and T2 reads value and T3 writes value to global
- Fault injection done by applying voltage dips to FPGA UT
  - Injected with 0.3Mhz frequency

Results(1/2)

- Five types of behaviors were reported with fault injections
  - System works normally
  - Generates different types of errors which were 100% detected
  - Generates different types of errors presents different fault detection capability
  - Crashes and needs to be reset
  - FPGA configuration failure
- Hw-S was able to detect 100% of transient faults injected for all benchmarks

TABLE III. FAULT COVERAGE ASSOCIATED TO BM1

System behavior	Voltage range [mV]	Voltage Dips [%]	Fault coverage [%]	
			RTOS	Hw-S
Behavior_1	[1200-956]	20.34	-	-
Behavior_2	[955-943]	21.42	100.00	100.0
Behavior_3	[942-858]	28.50	69.00	100.00
Behavior_4	[857-651]	45.75	0.00	100.00
Behavior_5	[650-0]	100.00	-	-

(a)

TABLE IV. FAULT COVERAGE ASSOCIATED TO BM2

System behavior	Voltage range [mV]	Voltage Dips [%]	Fault coverage [%]	
			RTOS	Hw-S
Behavior_1	[1200-1120]	6.67	-	-
Behavior_2	[1119-893]	25.58	100.00	100.0
Behavior_3	[892-858]	28.50	56.00	100.00
Behavior_4	[857-651]	45.75	0.00	100.00
Behavior_5	[650-0]	100.00	-	-

(b)

Results (2/2)

- After 15% dips in voltage RTOS can no longer detect
  - Due to loss of information because the voltage dip is too high
- Scheduling error most common
- Four behaviors were found:
  - Microp went to wrong task
  - Remained executing only one task
  - Same task was repeated
  - Same task was repeated and next task was skipped
- Hw-S performed more robust when exposed to voltage dips
- Area overhead was only 10.07%

Future works

- Hw-S
  - Voltage dips were only form of fault injection
  - Limited benchmarks
- ARINC 653
  - Health monitor is currently designed to only diagnosis system malfunctions and requires a ground maintenance crew for repairs
  - However, in a space environment the health monitoring services should have major adaptations and added complexity as human assistance is probably not available
  - No test data was presented in the use of AIR with RTEMS to demonstrate the systems reliability

References

Rufino, J., Filipe, S., Coutinho, M., Santos, S., and Windsor, J.
ARINC 653 Interface in RTEMS.
In *DASIA 2007 - Data Systems In Aerospace* (Aug. 2007), vol. 638 of *ESA Special Publication*, p. 26.

Tarrillo, J., Bolzani, L., and Vargas, F.
A hardware-scheduler for fault detection in rtos-based embedded systems.
In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on* (Aug 2009), pp. 341–347.