Integrating Redundancy Management and Real-time Services for Ultra Reliable Control Systems

Mohamed F. Younis Billy He
Honeywell International Inc.
Advanced Systems Technology Group
7000 Columbia Gateway Drive
Columbia, MD 21046, USA

Abstract

Integration of multiple real-time control modules has gained increased acceptance as a new trend in the industry during the past few years. For example, the avionics industry is embracing a new design approach referred to as Integrated Modular Avionics (IMA). The IMA approach encourages the use of general-purpose basic components and sharing of common resources to minimize the development and maintenance costs of avionics. However, the integration complicates the design and validation of these systems since sharing resources makes the behavior of the integrated application hard to predict and guarantee and therefore ensuring the fulfillment of timing constraints and maintaining fault-tolerance becomes a challenge. In this paper, we describe our experience with integrating redundancy management and real-time services in an IMA setup. The redundancy management system (RMS) masks faults through voting on the computation results from multiple redundant computing nodes and ensures synchronization among replicas. RMS is set to share the same CPU with real-time applications managed by a real-time operating system (RTOS). We discuss the issues related to that integration and our approach for addressing them. We describe validation efforts and summarize lessons learned.

1. Introduction

In recent years there has been a significant increase in the use of computers in embedded real-time control systems. Real-time applications such as factory automation, avionics and remote sensing are distinguished by the fact that their functional semantic is coupled with temporal correctness. Not only the embedded computer needs to perform the right control algorithm but it also needs to meet all the timing constraints associated with that algorithm. Some of the real-time applications are also safety-critical and require high level of reliability and fault-tolerance to ensure uninterrupted service that might risk the safety of the system. Avionics systems and nuclear reactors are good examples of such applications. Fault tolerance is usually achieved in such applications by the use of redundant components that are typically managed at the system level.

Integration of multiple real-time control modules has recently gained momentum in the industry. For example, the avionics industry is adopting a new design approach called Integrated Modular Avionics (IMA). The IMA approach encourages the use of general-purpose basic components and sharing of common resources to reduce development and maintenance costs of avionics. Resource sharing gives the control unit many benefits, such as reduced size and weight, and thus decreasing running costs of aircraft and space vehicles.

The Integrated Hazard Avoidance System (IHAS) and the Integrated Environmental Control System (IECS) are examples of IMA projects at Honeywell. The IHAS system integrates flight safety avionics such as Traffic Collision Avoidance System (TCAS), Enhanced Ground Proximity Warning System (EGPWS) and Weather Radar. The IECS system controls the operating environment to ensure safe use of the equipment on the aircraft and the comfort of passengers. For example, the IECS system adjusts (by cooling or heating) the operating temperature of hydraulic, electrical and mechanical power devices and equipment, de-ices and defogs windshield and controls cabin pressure and passengers' air-condition. The IHAS and IECS systems achieve substantial reduction in the very expensive flight-worthy hardware, in the weight and volume of avionics and in power consumption. Such reduction lowers development costs and increases the efficiency of aircraft operation.

However, the integration complicates the design and validation of these systems since sharing resources makes the behavior of the integrated application hard to predict and guarantee, and therefore ensuring the fulfillment of timing constraints and maintaining fault-tolerance becomes a challenge. For safety-critical real-time applications, it is necessary to be able to show, with a very high level of assurance, that a problem or failure in one application can be tolerated without disrupting the application. Thus, the IMA approach inherently requires strong partitioning among the different modules coexisting within the same system. Strong partitioning calls for well-defined boundaries among modules to ensure the continuity of operation in the presence of partial failure [1]. Containing effects of faults is crucial for the integrated environment to guarantee that a faulty component may not cause other components to fail and risk a total system failure.

In this paper, we describe our experience with integrating redundancy management with real-time services in an IMA setup. The redundancy management system

(RMS) masks faults through voting on the computation results from multiple redundant computing nodes and ensures synchronization among replicas. RMS is set to share the same CPU with real-time applications managed by a real-time operating system (RTOS). In the balance of this section, we introduce the fault tolerant architecture and its current implementation, integration issues and related work. In section 2, we discuss our approach for addressing the integration issues. We describe the implementation and performance measurements in section 3. Section 4 concludes the paper and summarizes the lessons learned.

1.1 System Architecture

Our IMA system is based on the Multi-computer Architecture for Fault-Tolerance (MAFT) [2], which is designed to provide extremely reliable computation in real-time control systems. The basic concept of MAFT, as depicted in figure 1, is to mask faults through voting on the computation results from multiple redundant computing nodes. The Redundancy Management System (RMS) is to detect, contain and tolerate the erroneous behavior resulting of a hardware or software fault regardless of the malice (benign or malicious), symmetry (symmetric or anti-symmetric) and the duration (permanent, transient or intermittent) of that fault. However, RMS does not cover generic faults in the application and in RMS' development. A mixed software and hardware implementation of RMS is used on the NASA X-33 prototype of the Venture-Star Reusable Launch Vehicle (RLV).

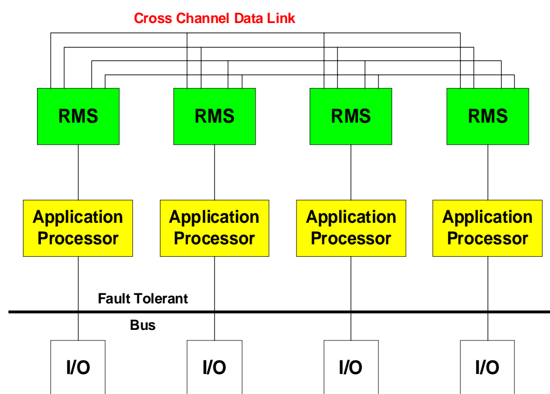

Figure 1: RMS coordinates among replicas in MAFT architecture

Using this architecture, every application module is executed multiple times simultaneously on different nodes (four in this example). RMS basically masks faults in the application data by excluding erroneous data and consistently providing correct data, from system-wide view, to replicated applications allowing faulty nodes to recover. RMS is available to the application processor (AP) as system service. The following are the main RMS functions:

System Synchronization: The computing platform is a distributed, loosely synchronized system with a fully connected communication network for all RMS units. Each RMS has its own clock and the system synchronization is achieved by exchanging the local time among all lanes and correcting the local clock according to the cardinality of clocks from all healthy RMS units. A distributed agreement mechanism is used to prevent any single point of failure and to protect the global system clock against any type of faults including Byzantine types.

Voting: Every application function will periodically send data to the associated RMS module via the direct communication links. Every RMS module will then send that data to all other RMS nodes through dedicated communication links, called Cross Channel Data Link (CCDL). After receiving all data copies, every RMS module will perform voting and send back the voted data values that will be used by the application for further computation. The voted data is used to mask the error generated by a faulty application node and allows maintaining consistency between the AP nodes. In addition, it assists in recovering from transient and intermittent faults by replacing any corrupted application data with voted values. Moreover, RMS votes on its internal state and error reports to maintain a consistent system-wide view of the system's health status.

Fault Tolerance: The ultimate goal of RMS is to prevent a system failure during a critical mission as a result of some error manifested by a fault on one node. The level of redundancy determines the fault-tolerance capability of RMS. A minimum of four nodes is needed to tolerate one Byzantine fault. By comparing the voted data values with the data submitted by the node, RMS detects errors and penalizes the faulty node. Nodes that exceed a certain penalty threshold will be excluded from voting, however voted data will continue to be send to the faulty node. Since all nodes will be using the voted data values, errors can be tolerated and the faulty node will get a chance to recover by using the voted data and hopefully can be re-admitted.

Current implementation of RMS consists of two parts: Fault-Tolerant Executive (FTE) implemented in software, and Cross-Channel Data Link (CCDL) for communication of voted data across multiple boards, implemented in hardware. On the X-33 vehicle, RMS is to run on a devoted CPU board and the application software is assigned to different CPU boards. Data are to be provided by the application software to the RMS for voting. Communication between the application boards and RMS is via VME backplane bus. The cross-channel communication links are electrically isolated so that one RMS node would not be affected with electrical faults on the another node. The use of dedicated board for RMS provides physical isolation from software application modules and thus ensures strong partitioning, yet in a costly way. With the increasing processing capacity of recent microprocessors and cost advantage of the integration, RMS is thought to share the processor and memory with the application module as we discuss next.

1.2 Integration Issues

The standalone RMS configuration can be expensive and inefficient, especially with the increased processing capabilities of modern technologies. An integration setup, figure 2, is thought in order to enhance resource utilization and reduce size and weight of RMS-based systems. The integration mainly avoids dedication of hardware resources for RMS. Instead, RMS would share these resources with the applications. Resource sharing between RMS and application software module challenges the system partitioning and triggers the following additional design issues related to system dependability and performance:

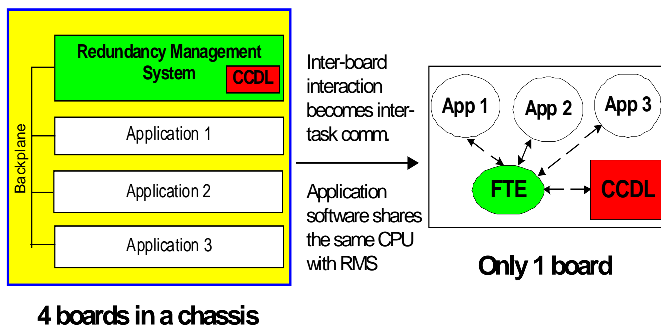

Figure 2: The integrated environment encourages resources sharing among RMS and application modules

Interaction with Application Software Environment: Running on RMS on a dedicated board provides RMS with autonomy of control of the hardware and prevents any conflict in permission to privilege operations. In the integrated setup privileged hardware access has to be assigned to either RMS or the real-time operating system, on which the development of application tasks is based.

Avoiding Thrashing Conditions: The present implementation of RMS is focused on functionality more than performance and throughput. By providing abundant resource capacities it is easier to validate RMS implementation compared to the integrated setup. In the integrated environment RMS shares the CPU with applications and thus CPU utilization is an issue. Therefore it is important to optimize RMS implementation in order to minimize the frequency of context switching and thus prevent any potential CPU thrashing.

Spatial Partitioning: Applications software would be allowed to share the same memory with the FTE component of RMS. Boundaries need to be enforced among the application modules and the FTE to ensure that the FTE memory space is protected against any application fault.

Temporal Partitioning: The CPU is interleaved among the FTE and application tasks. A robust scheduling mechanism needs to be deployed to protect the FTE shares of the CPU from application overrun or potential blocking conditions.

Clock synchronization: Since RMS is responsible for clock synchronization among replicas, task scheduling and other operating system services has to be aware of any potential clock adjustment. For example the application software environment has to use RMS reference clock to ensure consistency of time among the application and RMS tasks.

Handling of Voting Data: Application tasks submit data to RMS for voting and receive back reliable voted data consistent with other replicas in the system. In X-33 RMS implementation communication with RMS is possible only over a backplane bus. In the integrated environment, the application tasks and RMS reside on the same board and thus communication between them is local using typical inter-task communication (ITC) primitives. However, the integration would demand a reliable ITC mechanism that avoids blocking the progress of the FTE because of any application task.

In addition to the above technical issues, the integration approach has to have limited or even no impact on legacy RMS-based applications and their runtime environment. For example, it is not desirable to require changing the nature or the provider of real-time services to accommodate RMS in the integrated setup because it would necessitate a large scale revalidation for the application code, an effort that is very expensive in ultra reliable systems such as avionics. On the other hand it is imperative to minimize the changes to be made to RMS in order to limit the scope of testing and validation. Our approach for addressing the above issues is discussed in section 2. Prototype implementation and performance measurements are described in section 3.

1.3 Related Work

Many fault tolerant architectures have been developed since the early use of digital computers. The design approach varies based on the dependability and fault coverage requirements. A Historical perspective of the evolution of the fault tolerant architecture can be found in [3]. Architectures for mission-critical real-time applications were the focus of multiple research efforts, such as MARS [4], SIFT [5], MAFT [2], FTMP [6]. Most of these work address methodology and interface for the fault tolerance services, rather than integration issues with the application.

The Airplane Information Management System (AIMS) on the Boeing 777 commercial airplane is among the few examples of IMA based systems [7]. Although, the AIMS and other currently used IMA setup offer strong partitioning, they use special hardware and software environment and thus most of the integration issues we outlined were not faced.

2. Integration Approach

Multiple constraints had to be considered when addressing the integration issues mentioned in section 1. While it is expected to make changes to the implementation of RMS to fit the integrated environment, it is not acceptable to introduce new requirements or mandate the addition of new

features to the application software environment. Most of the application tasks are generally either developed using commercially available tools that are radically expensive to modify, or supplied by third party vendors who might not agree on adding new features. In addition it is desirable to minimize the scope of the changes made to RMS in order to limit the cost of revalidation. The following subsections describe how the integration issues were handled.

2.1 Software Architecture

A real-time operating system (RTOS) is typically used to schedule and manage application tasks. The RTOS has privileged access to hardware resources and all application-level tasks run in a non-privileged mode. On the other hand autonomous implementation of RMS on a dedicated board provides RMS with direct access and exclusive control of the hardware and thus prevents any conflict in permission to privilege operations. In the integrated setup privileged hardware access has to be assigned to either RMS or the RTOS, on which development of application tasks is based.

Interaction with Application Software Environment: Allowing RMS to maintain the privilege status requires RMS to manage application tasks and to provide RTOS typical services. Augmenting RMS features to act as a RTOS would complicate RMS design beyond acceptable level specially when considering legacy applications. Requiring particular real-time services to be provided by RMS would be viewed by application developer as a constraint that limits the usability of RMS, specially as the industry is moving away from proprietary software environments. In addition it is not cost-effective to compete with commercially available real-time operating systems.

Our approach, as depicted in figure 3, keeps the RTOS in the supervisor mode and runs the fault-tolerant executive of RMS as a service (daemon) task under the RTOS' control. Such approach would enable the integration of RMS with wide variety of real-time operating systems. The FTE can be defined as a system task if allowed by the RTOS or as a high priority user-level task. If the FTE is to be integrated as a high priority user-level task, it is very important to ensure that the FTE gets a priority higher than any other task. Since an application task can cause a priority inversion when it blocks while locking a shared device or data needed by a higher-priority task [8], the RTOS typically apply priority inherence protocol and temporarily elevate the priority of an application task. It is essential to ensure that no other task will compete with the FTE when it must run or preempt its execution.

This approach requires RMS to rely on the RTOS for the handling of the CCDL-related interrupts used to schedule internal RMS activities, and thus might delay the FTE invocation with the worst-case interrupt latency for that particular RTOS. Therefore, the FTE reactivation has to consider the worst-case interrupt latency. In order ensure

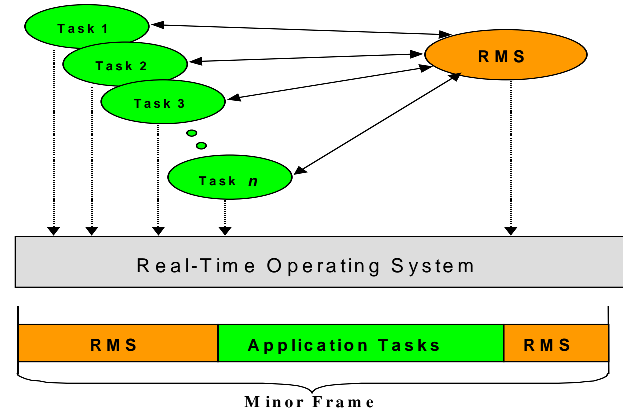

Figure 3: Integrated software environment

the FTE would be scheduled correctly, a timer has been added to the CCDL to track the idle duration between consecutive FTE operations. By programming the timer appropriately it is possible to ensure the activation of the FTE while accommodating for any RTOS latency in responding to the interrupt. The FTE code has been slightly modified to allow for a short duration of idle time in case of early activation.

Generally the use of interrupt in safety-critical hard real-time systems is totally avoided since it can jeopardize the system predictability. Typically the RTOS in such systems masks all hardware interrupts other than a clock-tick timer it use for tracking time for application tasks. Since it is not practical to stop the RTOS, specially a commercial one, from using the clock-tick timer during the FTE execution, the worst-case execution for clock-tick handling has to be considered while allocating time for the FTE. However it is not possible to allow clock-tick interrupts to take place while RMS is performing the clock synchronization among the replicas. Therefore, the non-interruptible portion of the RMS clock synchronizer module is extracted and invoked from the interrupt service routine to ensure that the clock-tick timer will not interrupt it. In our implementation, as described in section 3, we found that the CCDL-timer interrupt service routine takes less execution time than the worst-case interrupt latency of VxWorks and thus should not negatively impact the application timeliness.

Because in the federated setup RMS and application tasks communicate over an external communication device, buffering and communication delays limited the voting throughput. The integrated approach facilitates the communication of voted data between RMS and the application tasks. RTOS' inter-task communication primitives are to be used as later explained.

Avoiding Thrashing Conditions: In the stand-alone implementation, RMS has exclusive access to resources and thus correctness of the operation was the main focus while resource utilization, response time and throughput were

secondary issues. In the integrated environment RMS shares the CPU with applications and thus RMS' profile for CPU usage is an issue. Since using the CPU too often for short duration leads to excessive context switching and thus causing thrashing, it is important to optimize RMS' implementation by re-organizing its activities in order to increase the width of the periods in which RMS is idle and the application tasks seize the CPU.

After investigating the FTE operation, activities were classified into time-critical and flexible activities. For example, the fault-tolerant clock synchronization is time-critical and has to be performed at a specific slot in the frame to ensure that the system stays within the allowed clock jitter. Error logging and data voting are flexible as long as the application tasks do not miss their deadlines. Flexible activities were consolidated with time critical activities so that RMS does not have to run for many time-slots within the frame. In addition the integration allowed the elimination of double buffering the voted data since communication between RMS and application tasks is local. Section 3 reports the performance measurements.

2.2 Ensuring Strong Partitioning

In a federated RMS-application setup strong partitioning comes natural, although at a high cost. When RMS shares the same resources with the applications the system can be prone to wide failure if a task overruns its resource quota. Memory, CPU and CCDL are the only resources RMS use. The CCDL is designed to be memory-mapped and can be viewed as part of the RMS space. Therefore it is sufficient to enforce spatial and temporal partitioning in order to ensure the protection of RMS in the integrated setup.

Spatial Partitioning: Since applications code would be allowed to share the same memory with the FTE component of RMS, boundaries need to be enforced among the software modules and the FTE modules to ensure that the FTE memory space is protected against any software fault in the application software. Our approach relies on the memory management unit, commonly available on modern processors, to enforce memory partitions and protect the FTE space from faulty access by the application tasks.

To support the exchange of voting data between RMS and the application tasks special shared memory areas are to be allocated for the FTE. For each application task an outgoing buffer will be allocated in the task's address space with read-only access to other tasks including the FTE. In addition the FTE designates a per-task buffer for voted data in a shared memory area readable to the particular task. A lock-free approach is used to synchronize the execution of the FTE and application tasks, as later discussed.

Temporal Partitioning: Since the CPU is interleaved among the FTE and application tasks, a robust scheduling mechanism needs to be deployed to protect the FTE shares of the CPU from application overrun or potential blocking conditions. A timer is added to the CCDL to generate interrupts to the CPU urging the need to preempt the running application task and resume of RMS activities. Once the FTE finished the scheduled work it reprogram the timer and block waiting for a semaphore. The interrupt service routine would give the semaphore making the FTE ready to resume. Since most RTOS invokes the scheduler after serving an interrupt, assigning a high priority to the FTE would ensure on-time resumption. It should be noted that the strategy and algorithm for scheduling RMS and the application task is not discussed in this paper and the reader is referred to [9] for detailed the scheduling approach.

2.3 RMS-Application Synchronization

Data voting and clock synchronization among the replicas, are two of the main services RMS provides. Both services impact the scheduling of application tasks. Application tasks might block for the availability of the voted data. In addition RMS can introduce adjustments to the clock that can affect application-level events. Integrating RMS with the application introduces new issues related to scheduling tasks and the RTOS management of time, as we explain.

Clock synchronization: RMS performs frame-based clock synchronization among the replica [10]. Current clock values are exchanged and voted on. Each node adjusts either increments or decrements, its own clock to the voted clock in order to stay in sync with other replicas. Since RMS is responsible for clock synchronization among replicas, task scheduling and other operating system services has to be aware of any potential clock adjustment. Having different views of the frame boundaries would disturb the scheduling of RMS activities and hinders RMS' ability in maintaining clock synchrony. Therefore the software environment has to use RMS reference clock to ensure consistency of time between RMS and application.

Given that RMS does not run in privileged mode, the RTOS has to be informed about the adjustment in order to update the appropriate on-board hardware timer. Involving the RTOS in system's time adjustment complicates the integration and makes it more RTOS-dependent. Instead a 64-bit timer is added to the CCDL logic and used to derive the application scheduler. Because the timer is on the CCDL, it uses adjusted voted clock. By manipulating the RMS-application schedule in the CCDL-timer interrupt service routine, it is possible to enforce frame boundaries and ensure temporal partitioning between RMS and application tasks. In addition this approach can be easily applied to different real-time operating systems. The CCDL-timer is designed as memory-mapped device. At integration the address of the CCDL-timer is to be mapped for read-only access to all application tasks so that they can use RMS time as a reference. It is worth noting that the CCDL logic is implemented on a field programmable gate

array (FPGA) and thus introducing the timer did not require any changes to the CCDL schematics and layout.

Handling of Voting Data: Application tasks submit data to RMS for voting and receive back reliable voted data consistent with other replicas in the system. In autonomous RMS implementation communication with RMS is possible only over a bus and requires the data to be buffered twice at both ends. In the integrated environment, application tasks and RMS reside on the same board and thus they can interact locally using inter-task communication (ITC) primitives. While the integration provides an opportunity to optimize the voting performance, it can couple RMS with the application tasks and might require the revalidation of RMS when the application tasks are modified. Therefore, the integration would demand a reliable inter-task communication mechanism that avoids blocking the progress of the FTE because of any application task and makes the interface independent from ITC management.

Exchange of voting data between RMS and the application tasks would follow a protocol that never locks the FTE [11]. Each task that would submit data for voting has to request a shared memory area and demand the RTOS to make it readable to the FTE. A lock-free protocol is used to ensure that the FTE would be temporarily protected and would not block because of an application task. Avoidance of blocking the FTE ensures that RMS will continually provide its service on time even if some application tasks overrun. A task submitting data for voting will insert a time-stamp (T1) at the beginning of the data and another time-stamp (T2) at the end of the data. The time-stamp is simply the current frame number as read from the reference CCDL-timer. Given that the minor frame size is extremely larger than any clock adjustment made by RMS and that the 64-bit CCDL-timer will not reset to zero for millions of years, the time-stamp T2 will always be larger than or equal the time-stamp T1. Therefore in a race free condition if RMS read the data from the end to the beginning it will always find $T2 \geq T1$. If RMS access that data before the task finishes writing it, T2 will be less than T1 and RMS will ignore that data. The correctness of the semantics of such protocol is proved in [11]. Given such lock free mechanism the FTE has to check all tasks' voting data areas. In order to avoid voting on the same data multiple time in case the task is running at a slow rate, RMS would maintain the value of the time-stamp T2 of the most recent data voted for a particular shared area. In order for RMS to vote on a particular data, the value of the time-stamp T2 stored in the corresponding shared memory has to exceed the recent value that RMS has served.

On the other hand, the FTE would define a per-task shared data area, to which the RTOS will give the particular task read access. The FTE will include the voted data into that area for task to read. After a task submits some data for voting it can block for a semaphore (voted_data_ready), which the FTE gives upon completion of data voting. The application developer might decide to submit data for voting in batches so that the task does not have to block for long time. In this case multiple shared area per task will be used both by the task and the FTE. It should be noted that the FTE interface requires the size and format of the data to be defined at initialization time and to stay unchanged during the system operation.

3. Implementation Validation

To validate the integration approach a prototype has been built using commercially available components, as shown in figure 4. The prototype includes 3 VME backplanes that host six (two per backplane) PowerPC© 80 MHz processor modules. The software environment includes VxWorks©, a real-time operating system from WindRiver Systems Inc., and some of the application tasks from the X-33 vehicle management system. A version of RMS with the suggested changes has integrated within VxWorks.

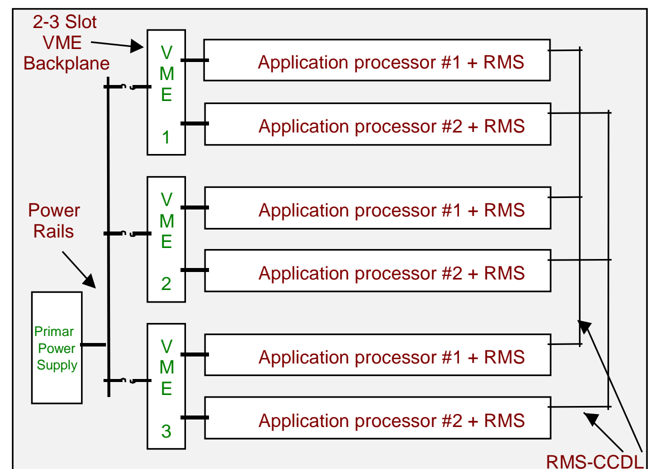

Figure 4: The Architecture for the Validation Prototype

A profiling tool called *WindView©*, supplied by *WindRiver Inc.,* was used to monitor the performance of the integrated setup. This tool enabled us to mark certain points in the source code as software events. As the test application was running on the target modules, each one of these events was time stamped with a resolution of 1 microsecond, and the readings were saved in a RAM file. The readings were later downloaded off-line to the host computer for analysis. The instrumentation of each event was associated with a 5-microsecond overhead that was taken into consideration while interpreting the results. Highly critical operations were not instrumented to avoid intrusion, instead time-stamps were taken before and after the operations. Three sets of voting data were considered, with the third set being the largest. Tables 1 and 2 summarize the time measurements for RMS activities in the stand-alone and integrated setup, respectively.

The integration allowed local communication between RMS and application tasks and thus enabled the removal of the data collection portion, which accounts for about 30%-60% of RMS execution time depending on the data size (large sizes usually take less bus transfer time per byte). In addition grouping multiple activities saved on the execution time, mainly due context switching. For example grouping the creation of error reports and the clearing the CCDL buffers saved about 35 μs, which represent about 15% reduction of the combined execution time of both actions. It should be noted that some activities were not affected.

Table 1: Performance of stand-alone RMS configuration

RMS activities	**Data set #1**	**Data set #2**	**Data set #3**
Frame Boundary Action	106	105	103
Clock Sync End Action	235	229	235
Application Synchronization	294	378	478
Error transmission & voting	241	242	243
Data collection & Transmission	2504	2537	2565
Data Voting	321	576	818
Create Error Report Action	159	156	157
Clear CCDL Action	63	62	61
Clock Sync Start Action	68	68	65
Total Exec Time in μs	***3991***	***4353***	***4725***

Table 2: RMS performance in the integrated setup

Combined RMS activities	**Data set #1**	**Data set #2**	**Data set #3**
Frame Boundary Action	107	103	104
Clock Sync End Action	235	229	235
App. synchronization, data transmission & Data Voting	1315	1950	2539
Error Report & Clear CCDL	186	184	189
Clock Sync Start Action	66	66	66
Total Exec Time in μs	***1909***	***2532***	***3133***

4. Conclusion

Although integration of safety-critical control has significant economic advantages, it raises many technical issues especially with redundancy management. In this paper we described an approach to combine real-time services with redundancy management for ultra-reliable systems. The integration enhanced resource utilization and increased voting throughput. The approach still maintains strong partitioning among the integrated applications and ensures the timeliness of critical redundancy management services. While few changes were introduced to RMS implementation, the impact on the application was minimal.

The approach is validated through prototype implementation using commercially available components. The implementation highlighted the need for better tools to help the developed tackle the complexity of testing fault-tolerant systems. In addition the internal design of the RTOS is found to have high impact on the complexity of the integration. RTOS' design centered on strong partitioning simplifies system's integration and validation.

References

[1] J. Rushby, "Partitioning in Avionics Architecture: Requirements, Mechanisms and Assurance," *Technical Report CR-1999-209347*, NASA, 1999.

[2] R. Kieckhafer, et. al., "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions on Computers*, Vol. 37, No. 4, pp. 398-405, April 1988.

[3] D. Siewiorek, "Architecture of Fault-Tolerant Computers: A Historical Perspective", *Proceedings of the IEEE*, Vol. 79, No. 12, December 1991.

[4] H. Kopetz, et. al. "Distributed fault-tolerant real-time systems: The MARS Approach," *IEEE Micro*, Vol. 9, No.1, pp. 25-40, February 1989.

[5] J. Wensley, et. al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proc. of the IEEE,* Vol. 66, No. 10, pp. 1240-1255, Oct. 1978.

[6] A. Hopkins, Jr., T. Smith, III, J. Lala, "FTMP--A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. of the IEEE,* Vol. 66, pp. 1240-1255, Oct. 1978.

[7] M. Johnson, "Boeing 777 Airplane Information Management System – Philosophy and Displays", in the *Proceedings of the Royal Aeronautical Society's Advanced Avionics Conference on Aq330/A340 and the Boeing 777 aircraft*, London, UK, November 1993.

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronisation," *IEEE Transactions on Computers*, pages 1175-1185, September 1990.

[9] Y.H. Lee, M. Younis, J. Zhou, "An Integrated Scheduling Mechanism for Fault Tolerant Modular Avionics Systems", *Proc. of the IEEE Aerospace Conference*, Aspen, Colorado, March 1998.

[10] P. Thambidurai, et al, "Clock synchronization in MAFT," Proc. IEEE 19th International Symposium on Fault-Tolerant Computing, 1989, pp 142-149.

[11] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 1, pp. 124--149, January 1991.