A Fault Observant Real-Time Embedded Design for Network-on-Chip Control Systems *

Christopher Zimmer
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
cjzimme2@ncsu.edu

Frank Mueller
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
mueller@cs.ncsu.edu

ABSTRACT

Performance and time to market requirements cause many real-time designers to consider components, off the shelf (COTS) for real-time systems. Massive multi-core embedded processors with network-on-chip (NoC) designs to facilitate core-to-core communication are becoming common in COTS. These architectures benefit real-time scheduling, but they also pose predictability challenges. In this work, we develop a framework for Fault Observant Real-Time Embedded design (Forte) that utilizes massive multi-core NoC designs to reduce overhead by up to an order of magnitude and to lower jitter in systems via utilizing message passing instead of shared memory as the means for intra-processor communication. Message passing, which is shown to improve the overall scalability of the system, is utilized as the basis for replication and task rejuvenation to improve fault resilience by orders of magnitude. To our knowledge, this work is the first to systematically map real-time tasks onto massive multi-core processors with support for fault tolerance that considers NoC effects on scalability on an actual massive multi-core hardware platform.

1. INTRODUCTION

ASIC-based real-time systems are costly to design in terms of time and money. Multi-core COTS processors are becoming increasingly used in the high-end handheld market and are also seeing increased use in the lower-end embedded control market. An example is the Freescale 8-core PowerPC P4080 that is being marketed in the power utility domain for control devices. In such processors, traditional software design techniques coupled with increasingly smaller transistor sizes can negatively affect the real-time predictability and the fault reliability. Predictability challenges in multi-cores are due to non-uniform memory latencies [1] as contention on buses and mesh interconnects increases.

Another trend is an increase in transient faults due to decreasing fabrication sizes. These faults surface as single event upsets (SEU) that can render computation incorrect. SEUs are faults that can modify logic or data in systems leading to incorrect computational results or software system corruption, which can result in temporary or even permanent incorrect actuator outputs in control systems if not countered. SEUs have three common causes: (1) Cosmic radiation, particularly during solar flares, (2) electric interference in harsh industrial environments (including high temperatures, such as in automotive control systems) and (3) ever smaller fabrication sizes and threshold voltages leading to increased probabilities of bit flips (for all) or cross-talk (for the latter) within CMOS circuitry [2, 3].

For example, the automotive industry has used temperature-hardened processors for control tasks around the engine block while space missions use radiation-hardened processors to avoid damage from solar radiation. An alternative approach is taken by commercial aviation. The latest planes [4] (Airbus 380 and Boeing 787) deploy off-the-shelf embedded processors without hardware protection against soft errors. Even though these planes are specifically designed to fly over the North Pole where radiation from space is more intense due to a thinner atmosphere, processors deployed on these aircraft lack error detecting/correcting capabilities. Hence, system developers have been asked to consider the effect of single-event upsets (SEUs), *i.e.*, infrequent single bit-flips, in their software design. In practice, future systems may have to sustain transient faults due to any of the above causes. COTS architectures are not specifically designed for real-time fault tolerant environments and contain few hardware-based fault tolerant mechanisms, such as processor radiation hardening. In previous work, researchers have designed techniques to mitigate SEUs in software using task scheduling [5, 6, 7]. This often leads to sophisticated scheduling techniques utilizing alternate algorithms, re-execution, or replication. In contrast, we argue that massive multi-cores with NoC interconnects greatly simplify scheduling while allowing for high levels of replication.

In a system with replication of entire task sets under the traditional shared-memory model, considerable strain is placed on memory controllers due to compounded memory pressure and coherence traffic resulting in contention. This contention limits scalability and reduces predictability of advanced multi-core architectures. In spite of the potential drawbacks, multi-core COTS processors remain quite attractive for real-time systems. For example, ARM promotes “dark silicon” each real-time task is mapped to a separate core as cores are plentiful [8]. Scheduling then amounts to simple core activation thereby eliminating context switching costs and preemption delays. Such an abstraction also facilitates parallel, replicated execution and voting in n-modular redundant environments to increase reliability.

*This work was supported in part by NSF grants CNS-0720496 and CNS-0905181

Contributions: This work introduces a Fault Observant Real-Time Embedded (Forte) design for large multi-core architectures with NoC interconnects. The detailed contributions of Forte are as follows: (1) Forte provides a task abstraction framework that takes advantage of *message passing* capabilities implicit in NoC systems to *eliminate the use of shared memory*. Forte thus *increases the overall predictability* of the system as contention on the memory controllers is reduced. Furthermore, Forte improves *scalability* for contemporary mesh-based NoC architectures. (2) Forte *improves reliability* by provisioning simultaneous task models of varying complexity and measuring the strength of association (coherency) to *facilitate voting* in a modular redundancy scheme. (3) Forte further ensures sustained reliability by enabling fine-grained task rejuvenation. This includes the ability to replace faulted data models and to refresh rejuvenated tasks to align redundant models. Such rejuvenation is critical particularly for long-running or 24/7 control systems. Experimental results with Forte show improvements up to an order of magnitude in overhead reduction over standard shared memory implementations, reduced jitter and scalability. Reliability is improved in line with results reported for modular redundancy. Yet, Forte sustains these reliability levels through rejuvenation as shown in experiments.

The remainder of this paper is structured as follows. Section 2 presents the design of our proposed framework. A case study developing an unmanned air vehicle control system is detailed in Sections 3 and 4. Section 5 provides the experimental framework. Section 6 presents experimental results. Related work is discussed in Section 7. The paper is summarized in Section 8.

2. FORTE DESIGN

This section provides an overview of the Forte framework to exploit massive multi-core processors to facilitate highly redundant real-time systems. Careful use of this technique can improve system integrity in the form of protection from soft errors by providing a framework for running multiple concurrent versions of a task, called **shadow tasks**, and verifying their output coherence. The framework assumes that each task is permanently assigned to a unique set of cores and that the number of tasks in the system is less than the number of cores. The scheduling system is periodic with dynamic priorities based on relative deadline. Figure 1 depicts our model of a massive multi-core NoC processor. Our sample processor model contains 64 processing elements connected in a mesh grid. Each processing element contains a switch so that network communication and routing can be handled without additional overhead to the processing pipeline. NoC processors often support both static and dynamic message routing. Due to this, our framework operates agnostic of the underlying message passing API.

Forte capitalizes on the additional processing elements available in advanced COTS processors to run multiple simultaneous system models. These models can vary in feature set and complexity, extending the model from the basic requirement, to a model with more precision/features to increase the system efficiency. We use the standard notation of ϕ, p, e, and d to denote phase, period, execution time, and deadline of a real-time task [9]. Using terminology from [10], we group the functional models and order them via complexity ranging from the most complex features to the "simple" baseline model. For example 1, consider two tasks $c =< \phi_c, p_c, e_c, d_c >$ and $s =< \phi_s, p_s, e_s, d_s >$, where c and s perform the same system function but c is a complex version of s, the baseline version. In Forte, we assert that $p_s = p_c$, since they

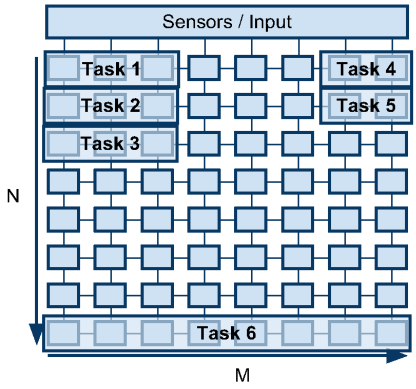

Figure 1: Forte Task Layout Over Cores

provide the same system function, though e_c may be larger than e_s. We further assert that s and c generate output data where a coherency range can be determined. For additional redundancy, we consider two more tasks, c_{shadow} and s_{shadow}. These tasks are added to the system as mirror images of c and s, operating on the same data to validate the correctness of each model's output data. To formalize the framework, we extend the classic task model such that:

$$\tau =< I, O, T, C, R > \quad (1)$$

- I is the set of inputs i for the m shadow tasks in T;
- O is the set of outputs o of τ that must be validated for coherence prior to allowing the output change on the system to take effect;
- T is the sequence of shadow tasks $< t_1, t_2, ... , t_m >$ where each element t_i in T is ordered by a descending complexity coefficient k_i such that $k_1 \geq k_2 \geq .. \geq k_m$;
- C is the set of coordinates $< x_1, y_1 >, < x_2, y_2 >$ that enable the system to bound τ to a specific core within the architecture;
- R is the set of data within a task that must be transferred to a rejuvenated task to ensure convergence. If natural convergence is used $R = \emptyset$.

The Forte framework characterizes each task within the system with a set of inputs and outputs. Figure 2 depicts a Forte task where the input phase splits the data so that three redundant tasks of f can operate on separate models of the input data in parallel. We use the term f to describe the defining function(job) of the real-time task. When a task finishes execution it then sends the outputs to a coherence check that determines the correct output for the system. These sets allow tasks to execute independently or to be chained together to facilitate data flow within the system. Figure 3 depicts how the various tasks communicate data without using shared memory. The model forms an abstract chain: Once a task generates output data, its output data becomes the input data for a subsequent task. This model can be implemented on NoC architectures through explicit message passing.

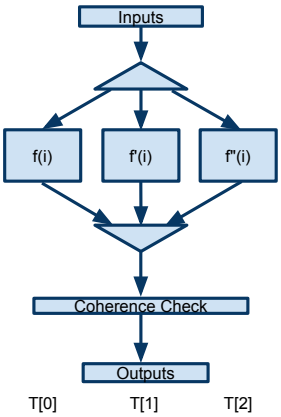

Figure 2: Abstract Task Layout After Splitting

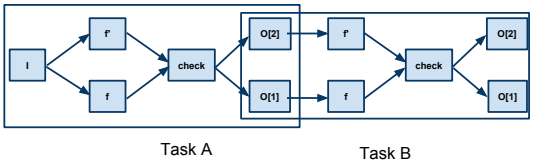

Figure 3: Task Chaining

2.1 Shadow Tasks

Forte is designed to exploit the high-level of concurrency that NoC architectures provide. Real-time systems deployed in harsh environments are subject to *Single Event Upsets (SEUs)*. These are compelling reasons to utilize the multi-core paradigm and generate several models of a single task called shadow tasks, which improves the level of data integrity of the system. In the previous example from this section, c,c_{shadow},s, and s_{shadow} are considered shadow tasks of a single system level task τ. In Forte, shadow tasks are represented in a complexity ordered list. To state this more formally, for each shadow task t_i in T, there is a complexity coefficient k_i, such that

$$< t_i, t_{i+1}, ..., t_m > \implies k_i \geq k_{i+1} \geq ... \geq k_m \qquad (2)$$

The complexity coefficient k_i is best generalized as a scoring value generated by deriving less precise real-time task models from the most sophisticated design. A degraded complexity model for real-time systems was put forth in [10]. In this work, a complex and a simple feature set for a given control task helped to increase the model safety. Deriving a score for k_i considers effects of a reduction in features as well as reductions in data precision or utilization for faster converging algorithms with a larger tolerance range ϵ.

2.2 Input

Real-time tasks have a variety of data models that can be supported in the Forte framework. Referring back to Figure 1, task 1 acquires input from sensors or other I/O devices that are not part of the task set. Task 2 derives input from task 1 and task 6 operates independently or receives input from a device that is pinned to the lower portion of the core layout. Supporting an abstract input set allows the framework to be flexibly used to deploy a variety of real-time

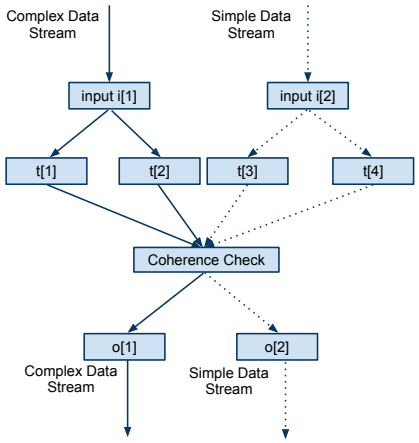

Figure 4: Data Stream Abstraction

tasks. Forte considers multiple data streams separated by complexity, shown in Figure 4. Streams enable shadow tasks of varied complexity to ensure that data is not unnecessarily losing precision by forcing a single stream of data.

In practice, input acquisition is a precondition for each task in Forte. If the input is derived from a sensor or other external hardware, it requires one of the shadow tasks to acquire the data and then distribute the data over the message passing network. If the input is derived from the output of another task, each of the shadow tasks must receive their input from a proceeding output of equivalent data complexity.

2.3 Output

Forte improves integrity by validating the coherency of each shadow task's data. A potential but undesirable result of this coherency validation is that the designer may have to reorder the code in control tasks to defer a decision until the shadow task decisions can be verified. Coherence formulations are determined by the system designer. Automatically identifying how to determine these is algorithm specific.

For a given task set, each shadow task operates on local data sets. Upon completing the necessary computation, the data is checked by the coherence-checking phase of the task. This may be performed by every shadow task or in a subset of them to reduce the data transfer cost. In the coherence check in Figure 4, shadow tasks $t[1]$ and $t[2]$ are of equal complexity and the data must match exactly. The same holds for $t[3]$ and $t[4]$. When this verification is complete, a range check is performed to validate that the data in the complex and simple streams are within a preset range. Certain features of the complex stream may not exist in a lower precision model. This makes it important to maintain multiple checks for each level of complexity. Successful coherency checks result in the mapping of output data to locations designated by complexity. This allows subsequent tasks dependent on this output to be mapped to the data of matching complexity. If the coherency checks fail, the failing task can be isolated to remove any impact it may have on the control system. If the failure is within the highest complexity model, subsequent shadow tasks that operate on that model can be canceled, allowing the system to rely on the less complex data models. If it is a lower complexity model that sustains the failure,

data of the higher complexity models can often be filtered to allow a lower complexity model to continue operation. This output data flow is shown in Figure 4. The result of the complex data stream is filtered into the simple data stream in this case. When using fine grained coherence checks in a n-modular redundancy configuration rejuvenation can be used to repair the faulting task.

The formalization of input and output sets also supports the handling of feedback control loops. Forte allows data within the output set to be specified in the input set of subsequent tasks. This formalization supports task chaining. Feedback loops are supported as a chained loop of multiple tasks or the redirection of a single task's output back into its own input.

2.4 TDMA

Contention can hinder performance on message-passing networks, *e.g.,* when multiple fault models transmit their data to coherence checks in the Forte system. Tasks could potentially overwhelm routers or their buffers with adverse affects on performance. Forte addresses this problem by arbitrating the underlying NoC network through Time Division Multiple Access(TDMA). TDMA makes Forte more predictable by reducing contention on the message-passing network and facilitating the bounding of worst case behavior for all message-passing phases.

2.5 Task Rejuvenation

Real-time control systems are developed to run for extended periods of time if not even 24/7. They may thus be exposed to multiple event failures over the course of their lifetime. Single event upsets are handled through coherence voting and elimination of the faulty data. A subsequent second or third event upset to one of the remaining redundant task may leave the system without decision capability as to which the correct results is. The objective of rejuvenation is to correct the faulting model to ensure that resilience of the model is sustained. According to a study from the high performance domain [11], as devices advance and die sizes decrease, the projected failures per hour for a single node in an HPC system is $4.1x10^{-7}$. Another study [12] from the satellite domain using a hardened COTS multi-core device evaluates the failure rate as $2.2x10^{-4}$ failures per hour. Both studies indicate that the probability of multiple-event upsets in a short time period is low. But if the runtime of the system is long, a second SEU is likely.

Forte addresses this challenge by supporting fine-grained rejuvenation as a part of the framework. Fine-grained coherence checks allow failing tasks to be identified. In Forte, an SEU is confined to a single task that is considered to have failed since tasks are associated with disjoint cores and do not share memory, i.e., only the failing task needs to be terminated. Subsequently, one of the remaining correct tasks supplies its output as input data to subsequent tasks of the terminated one during its rejuvenation. This is implemented as follows. The scheduler terminates the faulting task and creates a rejuvenated version of the task on the same core starting with newly initialized data values. The rejuvenated task is not caught up in its data output after such a restart and would fail the coherence check as thresholds would be exceeded. The coherence check is therefore temporarily relaxed to only validate the outputs of the remaining tasks (ignoring the rejuvenated one).

Coherence validation via voting is deferred until the rejuvenated task converges with the correct models in terms of its output. Many control algorithms exploit convergence algorithms in feedback loops to guarantee stability, i.e., they will naturally converge

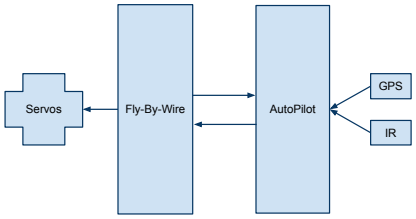

Figure 5: Paparazzi Design

over a period of time if the output is dependent on the input. In other cases, running state is maintained between each job invocation of a task so that models do not converge by itself. Here, the state of one of the remaining (correct) tasks is utilized to allow the rejuvenated task to catch up. Forte supports data refreshing of rejuvenated tasks as follows. A correct task is designated by the coherence module to refresh a rejuvenated task with local memory values specified during system design. These memory regions are transferred to the rejuvenated task in between job invocations to assure consistency. Data refresh is a requirement for non-converging algorithms. But it can (and often should) also be utilized to more quickly catch up with the correct tasks for converging algorithms. This reduces the vulnerability window to receive another SEU while operating under degraded redundancy (e.g., dual redundancy) during rejuvenation. After data refreshing (or convergence without refresh), the coherence validation can reactivate voting again upon reception of outputs from the reborn task within thresholds.

3. UAV APPLICATION

The next two sections describe our experimental implementation of the Forte design using a real control system. This section describes the control system and its tasks. The next section describes the changes necessary to move the control system into the Forte framework. To evaluate the design, we selected Paparazzi [13], a traditional shared memory real-time control system implementation. Paparazzi is an unmanned air vehicle(UAV) control software. We ported it using the Forte design framework and evaluated it on a hardware NoC architecture. Our port of the Paparazzi control system is based on a java implementation [14] that we rewrote in C++. Paparazzi is structured as two separate sets of real-time tasks that enable a switch between manual control of the aircraft and autopilot mode. These modes are detailed as Fly-By-Wire (FBW) and Autopilot (AP). The basic structure of Paparazzi allows only the FBW mode to control the servos. However, when there is no pulse position modulation (PPM) control, the autopilot mode sets the actuation by controlling the values that the FBW mode uses to control the servos. This relationship is detailed in Figure 5.

3.1 Paparazzi Autopilot-Base Design

The basic design of the shared memory version of Paparazzi uses several shared objects accessed various tasks to calculate vectors to control the UAV. This information consists of a navigator, estimator, and a flight plan. The following paragraphs will briefly cover each task and how it operates on these shared data structures in order to illustrate later how to redesign for a message-passing framework. The basic task layout for the auto pilot module with task dependencies and data flow are shown in Figure 6. **Navigation Task:** The navigation task is responsible for taking information from the GPS device, determining the current location of the UAV and then storing the values into the estimator data structure

for later tasks that cannot read the GPS data. It then compares this information against the flight plan and determines target metrics for the UAV to meet the flight plan. **Altitude Control Task:** The altitude control task is responsible for determining the control values to reach/maintain the desired UAV altitude. It first ensures that the system mode is set to allow autopilot control. It then obtains data from the estimator's z coordinates and determines the error from the desired altitude. It then uses this error factor to determine any corrections and commits them to one of the shared memory objects. **Climb Control Task:** The climb control task is responsible for determining the system's output in terms of thrust and pitch in order to maintain the necessary altitude. It takes as input the altitude determined in the altitude control task and the z directional speed vector determined in the navigation task. It uses these inputs to calculate the necessary pitch and thrust to control the altitude of the UAV's vertical changes. **Stabilization Control Task:** The stabilization control task uses data from the infrared (IR) device, the climb control task, and the navigation task. This task is responsible for determining the roll and any changes to the pitch. The stabilization control task in this implementation is also responsible for transferring the data to the FBW task that updates the actuation on the servos. The data sent is the pitch, roll, throttle, and gain to control the servos. **Radio Control Task:** This task takes the last radio control command from the FBW module and stores the data in the autopilot in case it needs to take over control.

3.2 Fly-By-Wire Base Design

The Fly-By-Wire(FBW) task set is used to control the servos and to take control from the ground control unit, the latter of which is not exercised in this implementation. The task layout of the FBW module is shown in Figure 7. **Pulse Position Modulation (PPM):** The PPM task receives the radio commands from the PPM device and uses them to control the servos of the UAV if the autopilot mode is not enabled. **Transfer to Autopilot:** This task takes the message retrieved from the PPM device and transfers it over the systems designated bus to the Radio Control Task. **Check Fail Safe Mode:** This task controls whether the auto pilot or the PPM device is controlling the UAV. It validates several device-based metrics to determine if the device is still receiving signals from the PPM device or if a fail-safe mode has been activated. **Check Auto Pilot:** This task controls the servos based on data received from the AP. The task receives data from the stabilization control task over the systems specified bus and then transfers these control values to the servos for actuation. **Flight Model and Simulated Devices:** In order to function appropriately Paparazzi requires a GPS device, IR device, and a functional flight model. The Flight model specifies flight dynamics based on the rudimentary version found in the Paparazzi open source code. The GPS device infers several metrics based on its current position, its last position and the change in time. The IR simulates a dual axis differential IR device, that uses IR temperature readings between space and the earth to stabilize the roll and pitch of the aircraft. The output data from the IR device is critical in the stabilization task.

4. FORTE IMPLEMENTATION

4.1 Input and Output Tasks

Implementing Paparazzi using the Forte design required analyzing the shared memory accesses that occurred within the task set and expressing them as data-flow relationships between tasks. The original implementation of Paparazzi uses logical objects to store data in containers. This eased programming requirements in that it made the data logically organized. However, it also made all

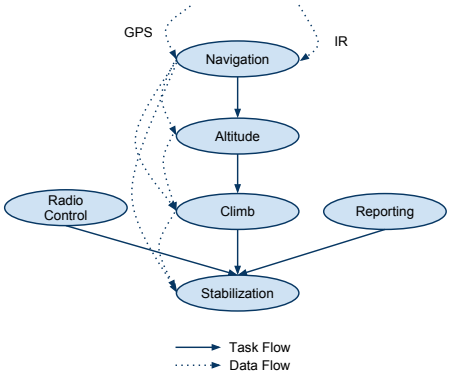

Figure 6: Auto Pilot Task and Data Flow

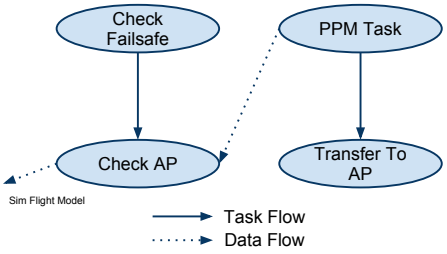

Figure 7: Fly-By-Wire Task and Data Flow

data in these objects globally accessible. While this is suitable for single-core implementations, using shared data in multi-core scenarios adds overhead. We remedied this by transforming data flow relationships to remove shared object containers altogether. They were replaced by data designated in two ways.

First, we utilize local data when data is only operated on within a task. The majority of data in our implementation could be categorized as local data. This contains all temporary variables and most of the state variables that update the primary flight metrics during operation.

Second, we utilize remote data. This data is stored locally but the actual data values originated else-where and are communicated between cores via sends and receives. Remote data values are written to local memory of the task before the task is released. In Figure 6, the dotted lines represent the flow of remote data in the auto pilot module.

We then converted each task into Forte tasks. Each Forte task consists of an input phase, a computation phase, and an output phase. The input phase of each task is generic. The task simply receives data and stores it in local memory for subsequent execution. Task computation differs from the shared memory version only in that instead of operating on global containers all data is local to the tasks core. The output phase sends any data to subsequent tasks according to the data flow specifications.

4.2 Scheduler

In the introduction of this paper, we made the claim that massive multi-core architectures could ease the problem of task scheduling. Trends in the market indicate that in the near future architectures with tens if not hundreds of cores will be arriving. In the

past, processing resources were in heavy contention and sophisticated scheduling techniques were needed to arbitrate access to limited resources. The term limited can no longer be used to describe processing resources for massive multi-core architectures. For the Forte implementation of Paparazzi, the scheduler is a simple periodic scheduler. The scheduler statically deploys each task to its own core where it remains stationary. Taking advantage of the massive multi-core architecture, no tasks shared a core. Scheduling thus reduces to core activation/deactivation to release or terminate a task. Each task would then be set to sleep until it received a NoC-based message from the scheduler core waking it up to perform its task. The impact of the sleep state is significant in terms of power consumption. As the number of cores on these architectures scales up, that ability to power them simultaneously will become a serious challenge. In order to limit the scope of the power consumption of such chips, many chip designers are implementing low power sleep modes with instant-on functionality. This enables software to constantly turn off and on the resources needed while conserving power.

4.3 Fault Models

To simplify our experimental implementation we integrated an n-modular redundancy configuration using the Forte model instead of a Simplex implementation. In our evaluation, we use a triple modular redundancy model. This shows the flexibility of the architecture in that we are able to use the Fortes design to have three completely simultaneous instances of Paparazzi. This enables coherence checks to identify the faulty model in times of failure so that voting can occur to determine which model controls the simulated servos.

4.4 Coherence Checks

We designed several coherence checks to enable robust fault checking for our Paparazzi implementation. Since our fault model in Forte was designed with redundancy tasks, our coherence checks simply verify data consistency. Each coherence check is designed as a sporadic task that immediately follows the execution of a system task in the Paparazzi suite using precedence constraints. Each coherence task is assigned to a specific system core. When the coherence task receives data from the first model, it sets a timeout in order to not wait indefinitely for the remaining models to transmit their data. When all of the models have transmitted the data, the coherence check validates the data. When there is a validation error, the coherence check uses a $2/3$ majority. It determines the failing model and notifies the voting routines to prevent the faulting model from controlling the system servos. When a timeout occurs coherence is checked between the models that did submit data, any models that did not submit data are considered to have failed.

4.5 Rejuvenation

Rejuvenation is implemented in Forte in two ways. The feedback control algorithms support natural convergence and, as such, just require a restart mechanism and a warm up phase to re-enable coherence validation. Paparazzi utilizes such natural convergence, i.e., our implementation exploits this restart capability. In addition, rejuvenation with refreshed data was realized as an optional extension. This allow us to compare the time (overhead) for convergence with and without refresh. To facilitate rejuvenation under data refresh, the coherence module uses the message passing network to indicate the source data refresh, i.e., one of the remaining correct tasks (cores). Refresh data is transmitted during the next idle phase to ensure non-interference with real-time deadlines of the correct

	0	1	2	3	4	5	6	7
0	OS	Scheduler	FM Sim 1	FM GPS 1	FM IR 1	Nav 1	Alt 1	CC 1
1	Stab 1	Rad 1	Report 1	Fail Safe 1	Send To AP 1	Check AP 1	PPM 1	Co - Check
2	FM Sim 2	FM GPS 2	FM IR 2	Nav 2	Alt 2	CC 2	Stab 2	Rad 2
3	Report 2	Fail Safe 2	Send to AP 2	Check AP 2	PPM 2	FM Sim 3	FM GPS 3	FM IR 3
4	Nav 3	Alt 3	CC 3	Stab 3	Rad 3	Report 3	Fail Safe 3	Send to AP 3
5	Check AP 3	PPM 3						
6								
7								

Figure 8: Paparazzi Task Layout

tasks. The refresh data is also received during the idle phase of the restarted task as redundant tasks are harmonic (not only in period but also in idle phase). Received data subsequently refreshes uninitialized state in the tasks, either to ensure that outputs are within coherence thresholds or, as given in the Paparazzi example, to speed up convergence amongst the redundant tasks.

5. EXPERIMENTAL FRAMEWORK

Our experiments were conducted on a Tilera TilePro64 development board. This platform features a 64 tile (core) chip multiprocessor (CMP) suitable for the embedded space with lower power requirements [15]. The Tilera platform has been selected for satellite deployment. Tilera processors support both message-passing and coherent shared memory models, and the choice is up to the user. Tiles are connected by multiple meshed NoCs that support memory, user, I/O, and coherence traffic on separate interconnects. Each tile processor is equipped with level 1 caches and split TLB making each core a fully independent processor. For evaluating our framework we, implemented the PapaBench real-time task set from the Paparazzi UAV project. Two implementations were created for evaluating not only the framework's fault resilience but to also compare computational jitter in systems relying on shared memory vs. message-passing. The shared memory task sets follow the proposed model in the paper (but with input and output phases integrated with computation phases of tasks). Figure 8 depicts the system layout. The figure illustrates the linear task layout across the tiles. This layout is agnostic to the execution models (shared memory vs. message-passing). All experiments using more than two tasks arbitrate access to the NoC using TDMA as described in previous sections. This reduces the impact of NoC effects on the system.

We conducted experiments with both the message-passing and shared-memory approaches using triple concurrent redundancy to evaluate the effectiveness of the Forte framework. We employed targeted fault injection in each of the models by generating data errors to evaluate the effectiveness of the coherency checks. To model full redundancy, we duplicated the simulated UAV hardware so that each model operated off of unique device inputs.

6. EXPERIMENTAL RESULTS

Table 1 depicts the number of injected faults that are detectable (resulting in output faults) and the number of actually recognized faults. The results indicate that all detectable faults were recognized and subsequently averted using voting in the coherence checks. We implemented a single coherence check to validate sys-

SEU Type	Detectable SEU Count	Recognized
Heap Flip	15	15
Device Failure	3	3
Stack Flip	10	10
Read Only Flip	4	4

Table 1: Fault Injection Evaluation

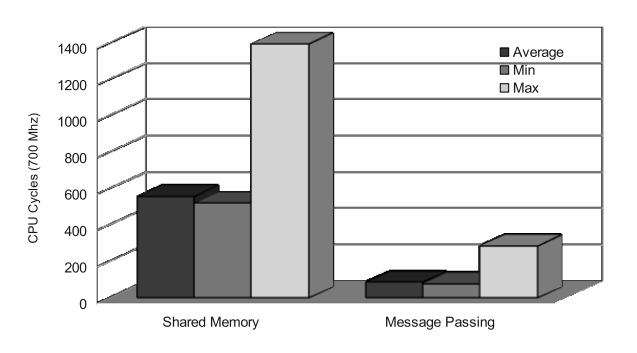

Figure 9: Overhead of Coherence: Shared Memory vs. Message Passing

tem data prior to servo actuation. The coherence check assessed the output data that was passed over the peripheral bus to the servo controller. We only included outcomes from SEUs that created an actual effect on the output state of the running systems. Faults were categorized as follows: (1) Downstream data errors: prior to servo actuation, outputs of the models were compared for consistency. By using three duplicated models, the faulting model is defeated (voted out). (2) Read-only (RO) memory upsets caused one of the models to fault. When this occurred, one model failed the coherence check through a timeout mechanism set by the coherence check's data deadline.

The next experiment exemplifies one of the major benefits of the message passing design over shared memory. Figure 9 depicts the computational cost (in cycles) for accesses to data subject to coherency checks for both models. These results measure the coherence within the climb control model that maintains computational control over five of the system control variables. This coherency check validates the consistency of the three simultaneous climb control data sets. As Figure 9 indicates, shared memory results in an order of magnitude performance penalty compared to message-passing. The overhead of the latter is due to maintaining coherency for remote writes for the validation checks. The message-passing model eliminates the need for coherence and reduces conflicts on the interconnects resulting in more predictable and lower execution time.

Figure 10 depicts the overheads for computing integer data in the climb control task. These results show stable timings for task computation with message passing, much in contrast to shared memory. We evaluated integer computations because of a lack of hardware floating point units (FPU) on the Tilepro64. This data demonstrates how easily contention on the NoC results in jitter. In this result, three simultaneous models are executing while the previous results utilized only one active tile during the actual check. Note that when multiple tiles are active simultaneous jitter is easily introduced into shared memory accesses. In contrast, TDMA arbitrates NoC access for messages.

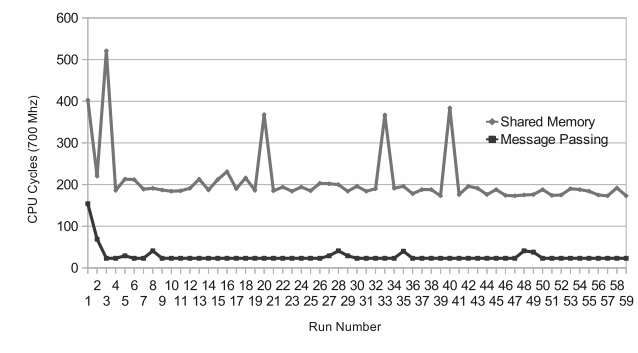

Figure 10: Climb Control Task Jitter: Shared Memory vs. Message Passing

SEU Scheme	Time To Repair	Mean Time to Failure
No Rejuvenation	∞	157 Days
Natural Convergence	8 (2s)	$2.27x10^8$ Days
Data Driven	1 (250ms)	$2.05x10^9$ Days

Table 2: Rejuvenation: Time to Full Restart

We implemented a naturally converging model and a data refresh model to assess the benefits of rejuvenation. To compare the models, it is necessary to measure the time from failure until triple redundancy is restored, i.e., voting within the system can restart. Table 2 depicts this as the time to repair for each scheme. Column two indicates that natural convergence took eight job cycles (periods) before voting could restart while rejuvenation with data refresh was able to accurately measure coherence one job (period) after the original failure. Column three assesses the mean time to failure (MTTF) for each scheme. Without rejuvenation, the model to derive data for the second row follows the standard MTTF calculation $MTTF_{TMR} = 5/(6\lambda)$. The model with repair via rejuvenation used to derive results for the third and fourth rows is based on a modified Markov formulation that calculates MTTF as $MTTF_{TMR-Repair} = 5/(6\lambda) + \mu/(6\lambda^2)$ [16]. μ is the maximum number of repairs that can be performed within an hour. We evaluated our model based on the λ derived from a radiation-hardened Tilera processor for these results [12]. This provides a worst-case λ as the processor is hardened and the error rates are evaluated in space making λ higher than values derived for single-node failures for terrestrial applications, e.g., λ values reported for HPC environments. As can be seen from the results, rejuvenation increases reliable operation by six to seven orders of magnitude.

The experiments thus far assess the cost of communication in a real-time system that only exercises some aspects of Forte's design. To evaluate the limits of Forte, we implemented a micro-benchmark that transfers a data payload of varied size between two cores. The benchmark utilizes both shared memory and message passing to evaluate the cost of aggregate data transfers. Memory addresses are uniformly distributed across the L3 cache by the hardware. Notice that this is a virtual L3 cache implemented through a hypervisor by distributing memory references over the L2 caches of all cores. The distribution uses a home-based protocol where the hash of a shared memory address redirects a look-up to a home core over a specific coherence interconnect on the NoC. Hashing can thus significantly increase the performance of shared memory by reducing the average distance to cached data and by increasing cache capacity of L3 to the aggregate of all L2 caches. This

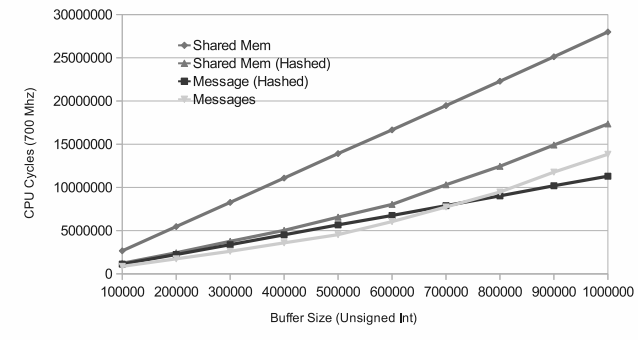

Figure 11: Bulk Transfer Overhead: Shared Memory vs. Message Passing

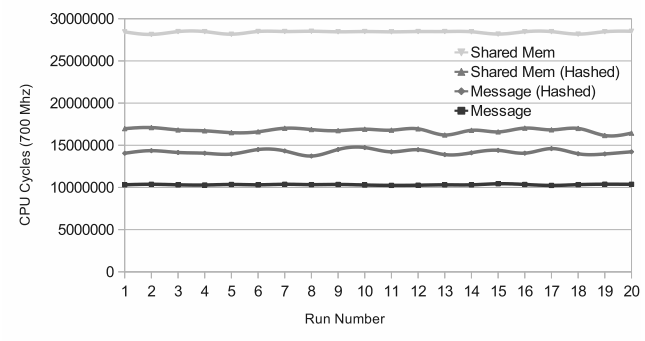

Figure 12: Bulk Transfer Jitter: Shared Memory vs. Message Passing

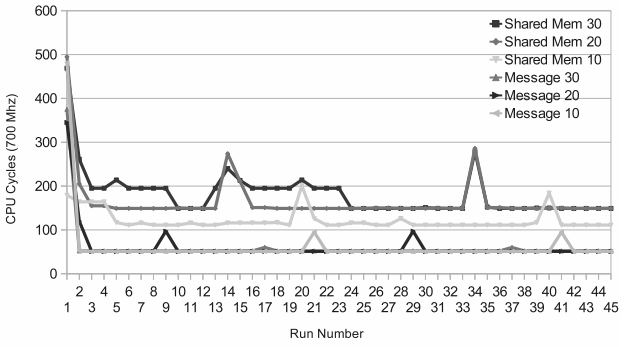

Figure 13: Scaling Contention: Shared Memory vs. Message Passing

effect is demonstrated in Figure 11. Hashed shared memory significantly outperforms the non-hashed counterpart. However, even with a significant reduction in the cost of shared memory access, the message passing transfer outperforms shared memory in both configurations. Figure 12 depicts the cost (in cycles) of a zero-contention data transfer scenario over multiple runs illustrating the jitter for the respective models. These results prompted us to not use hashing in the previous experiments. The results indicate that hashing improves performance of shared memory but at the cost of 1.5% additional jitter since accesses to distributed L3 have variable hop counts over the NoC. A jitter of 1.8% is even observed in the message passing results when L3/hashing is active as a result of the forced address resolution and non-local data placement associate with hashing. Message passing under deactivated virtual L3/hashing results in lower jitter(only .5%).

Next, we evaluated the scalability of the Forte design. We ran a single Paparazzi model of the full system in this experiment. The number of replicas of the altitude control task was scaled up gradually from 10 over 20 to 30 redundant instances. All replicas were executed in parallel on separate cores. This raised the overall utilization to 45 cores for the Paparazzi task set including the scheduler and coherence check. Figure 13 depicts the cost of data transfer/computation (in cycles) over multiple benchmark run for 10, 20 and 30 replica. A relatively inconsistent access cost is incurred with 30 replica cores for shared memory. Interestingly, a consistent additional overhead of approximately 50 cycles is observed for shared memory using 20 and 30 replica cores relative to just 10 cores, which can be accounted to scalability limits of the coherence protocol due to contention on the coherence interconnect. In contrast, additional replicas have virtually no measurable effect on the overheads for message passing (without L3/hashing) as TDMA arbitrates NoC access when messages are transferred. The occasional spikes in these results are caused by the virtualization layer in our experimental platform, which periodically activates a required monitoring daemon resulting in system noise. Such daemons would need to be eliminated or modeled as a separate task to meet real-time requirements.

Overall, the results indicate superior performance, increased predictability and reduced jitter of pure message passing (without any background coherence protocol) in this massive multi-core platform with a mesh-based NoC. Performance and predictability benefits of message passing over shared memory improve as the number of utilized cores increases, *i.e,* message passing scales in contrast to shared memory programming. The cause of these benefits lie in the potential of one-sided communication and TDMA arbitration of message passing in a push-based (explicit) access model. These advantages cannot be matched shared memory protocols with its pull-based (implicit) on-demand access requests and its required hand-shake semantics of the coherence protocol.

7. RELATED WORK

There is significant related work in the area of fault tolerance. Past approaches utilize scheduling, replication, or radiation hardening to achieve fault tolerance. Scheduling techniques, such as in [5, 6, 7], often introduce sophisticated scheduling policies to track faults. In particular, [5] introduces a last chance scheduling technique with the notion of task alternates to correct data in times of faults. A complicated scheduling algorithm then delays the execution of these alternates until the last possible moment to provide a fault tolerant schedule. We use advanced multi-core architectures to remove the need for such sophisticated scheduling algorithms by enabling the software to run alternates simultaneously at virtually no additional resource cost.

There exists a significant amount of work on detection of and protection against transient faults. Hardware can protect and even correct transient faults at the cost of redundant circuits [17, 18, 19, 20] Software approaches can also protect/correct these faults, *e.g.*, by instruction duplication or algorithmic design [21, 22, 23, 24, 25] Recent work focuses on a hybrid solution of both hardware and software support to counter transient faults [26, 27, 28]. Such hybrid solutions aim at a reduced cost of protection, *i.e.*, cost in

terms of extra die size, performance penalty and increased code size. Hybrid approaches have been proposed for selectively protecting hardware regions, for control-flow checking and for reduced instruction and data duplication in software [26]. Data representations, however, have been widely ignored. Radiation hardening is another common technique in fault protection for real-time systems [29, 30] with overheads in costs and speed. In contrast to our work, these solutions either promote hardware approaches or do not consider massive multi-cores (or even real-time systems).

Modular redundancy is a replication technique[31]. This work provides an easy to implement and validate approach to ensuring fault tolerance. The technique has been used widely in research. [32] describes a heterogeneous NoC architecture to implement triple modular redundancy. This work focuses on a specialized architecture that supports multiple levels of hardware integrated fault detection. This work uses TDMA on a NoC to interconnect the various IP elements in the architecture. Our work also utilizes a replicated task mapping but differs in that it is a pure software approach that enables comparisons of varying task complexity models with COTS applicability.

Rejuvenation [33, 34] is a technique originally introduced as a software restart technique to protect long-running software. Rejuvenation is often associated with rebooting. A major hurdle in software rejuvenation is data loss due to the rejuvenation. Forte uses software rejuvenation to maintain reliability in the control system. Data loss is circumvented through selective rejuvenation and data refreshing from validated data models.

8. CONCLUSION

We have presented the design of Forte, a framework that utilizes massive multi-core NoC architectures in order to create a reduced jitter and fault tolerant real-time environment. The primary tenets of this approach encompassed systematic restructuring of traditional real-time tasks to eliminate the use of shared memory by instead relying on message passing to move data between tasks. By reducing contention on memory controllers, it becomes more feasible to scale up the number of cores while sustaining performance and predictability. This enables support for fault tolerance through replicated real-time tasks combined with consistency verification and task rejuvenation using modular redundancy. Our results feature experiments with triple modular on-chip redundancy for a UAV control system and illustrate capabilities of Forte to detect errors and correct tainted results due to data errors, such as SEUs. We also show that by putting greater emphasis on message passing and eliminating shared memory accesses, we are able to increase predictability and decrease overheads by up to an order of magnitude. System reliability can be further increased by six to seven orders of magnitude when triple modular redundancy is combined with naturally converging and refresh-assisted rejuvenation, respectively.

9. REFERENCES

[1] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of distributed shared memory architectures for noc-based multiprocessors," *Journal of Systems Architecture*, vol. 53, no. 10, pp. 719 – 732, 2007, embedded Computer Systems: Architectures, Modeling, and Simulation.

[2] C. Constantinescu, "Trends and challenges in vlsi circuits reliability," *IEEE Micro*, pp. 14–19, July-August, 1996.

[3] V.Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *IEEE Computer magazine*, pp. 106–108, January, 2006.

[4] M. Pignol, "Cots-based applications in space avionics," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, march 2010, pp. 1213 –1219.

[5] C.-C. Han, K. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *Computers, IEEE Transactions on*, vol. 52, no. 3, pp. 362 – 372, Mar. 2003.

[6] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 272 –284, Mar. 1997.

[7] M. Cirinei, E. Bini, G. Lipari, and A. Ferrari, "A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, Mar. 2007, pp. 1 –8.

[8] J. Donovan, "Arm cto warns of dark silicon." [Online]. Available: http://www.eetimes.com/electronics-news/4136890/ARM-CTO-warns-of-dark-silicon

[9] J. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[10] J. G. Rivera, A. A. Danylyszyn, C. B. Weinstock, L. Sha, and M. J. Gagliardi, "An Architectural Description of the Simplex Architecture," Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania, Technical Report, 1996.

[11] J. T. Daly, "Running applications successfully at extreme scale: What is needed?" ASCR Computer Science Research, Principal Investigators Meeting, ANL-TR LA-UR-09-1800, 2008.

[12] M. Cabanas-Holmen, E. H. Cannon, C. Neathery, R. Brees, B. Buchanan, A. Amort, and A. Kleinosowski, "Maestro processor single event error analysis." [Online]. Available: http://www.aero.org/conferences/mrqw/documents/09/21_Manuel_ Cabanas-Holmen_ MRQW09.pdf

[13] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "Papabench: a free real-time benchmark," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, F. Mueller, Ed. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/678

[14] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl, "Exhaustive testing of safety critical java," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '10. New York, NY, USA: ACM, 2010, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/1850771.1850794

[15] "Tilera processor family," http://www.tilera.com/products/processors.php.

[16] C. Singh, "Reliability modeling of tmr computer systems with repair and common mode failures," *Microelectronics and Reliability*, vol. 21, no. 2, pp. 259 – 262, 1981.

[17] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3ghz fifth generation sparc64 microprocessor," in *Design Automation Conference*. New York, NY, USA: ACM Press, 2003, pp. 702–705.

[18] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications*

Conference. Proceedings, vol. 1, 1996, pp. 293–307.

[19] Y. C. B. Yeh, "Design considerations in boeing 777 fly-by-wire computers," in *IEEE International High-Assurance Systems Engineering Symposium*, 1998, p. 64. [Online]. Available: http://www.computer.org/proceedings/hase/9221/92210064abs.htm

[20] J. R. Sklaroff, "Redundancy management technique for space shuttle computers," *IBM Journal of Research and Development*, vol. 20, no. 1, pp. 20–28, 1976.

[21] P. Shirvani, N. Saxena, and E. McCluskey, "Software-implemented edac protection against seus," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 273–284, 2000.

[22] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 35–44. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SCAM.2001.972664

[23] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[24] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *International On-Line Testing Symposium*, 2003, pp. 137–143.

[25] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CGO.2005.34

[27] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *International Symposium on Computer Architecture*, 2005, pp. 148–159.

[28] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *International Conference on Embedded Software*, 2005, pp. 203–209. [Online]. Available: http://doi.acm.org/10.1145/1086266

[29] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng, "Analysis and optimization of fault-tolerant embedded systems with hardened processors," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, Apr. 2009, pp. 682 –687.

[30] I. Troxel, E. Grobelny, G. Cieslewski, J. Curreri, M. Fischer, and A. D. George, "Reliable management services for cotsbased space systems and applications," in *Proc. International Conference on Embedded Systems and Applications (ESA), Las Vegas, NV*, 2006.

[31] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200 –209, Apr. 1962.

[32] R. Obermaisser, H. Kraut, and C. Salloum, "A transient-resilient system-on-a-chip architecture with support for on-chip and off-chip tmr," in *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, May 2008, pp. 123 –134.

[33] W. Yurcik and D. Doss, "Achieving fault-tolerant software with rejuvenation and reconfiguration," *Software, IEEE*, vol. 18, no. 4, pp. 48 –52, 2001.

[34] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: analysis, module and applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, Jun. 1995, pp. 381 –390.