n-Modular Redundant Real-Time Middleware: Design and Implementation

James Marshall, Gabriel Parmer, Rahul Simha
{jcmarsh, gparmer, simha}@gwu.edu
The George Washington University

Gedare Bloom
gedare.bloom@howard.edu
Howard University

ABSTRACT

Cyber-physical systems (CPSs) have stringent requirements for size, power, cost, and reaction time. Fault protection mechanisms negatively impact some or all of these considerations. We introduce a software-only framework that leverages the modular and robust design of CPSs to allow more flexibility in detecting and recovering from transient faults. We describe an implementation of this framework that adheres to the POSIX specification, with the exceptions of architecture specific timers, CPU pinning, and a mechanism to walk the pages of a process. The system is evaluated with a set of micro-benchmarks, simulated failure injection campaign, and with simulated tasks performed by an example CPS.

1. INTRODUCTION

Transient faults in cyber-physical systems (CPS) such as satellites may be caused by electrical noise [20] and ionizing radiation [10]. Also known as single event upsets (SEUs), these faults manifest as "bit-flips" and may cause data corruption, computational errors, and execution faults. Hardware based double and triple modular redundancy (DMR and TMR) [11] are the primary mechanisms for detection and recovery. Specialized hardware can protect against some sources of transient faults, but are more expensive than non-redundant commodity hardware and require more power and space.

Small, standardized satellites such as CubeSats [7] are becoming increasingly popular [23]. CubeSats, designated by the number of 10^3cm "units" of volume that they occupy (eg. a 6U CubeSat measures 10cm x 20cm x 30cm), have a more limited budget in terms of cost, volume, and power than traditional satellites. CubeSats typically rely on hardware watchdog timers to power cycle commodity processors in case of an SEU induced lock up [17]. Software-based radiation mitigation has the potential to improve fault tolerance for CubeSats and other non-critical space applications that can not afford hardware-based solutions.

We present Scalable System Support for Reliable Embedded Software (S^3RES), a framework for component-based systems that detects and recovers from faults with software-based redundancy that may be scaled on a per-component basis from TMR, DMR, to no redundancy. S^3RES is a user-space approach that uses POSIX interfaces as much as possible to replicate, monitor, and replace components. The unit of validation is a *component*, as defined by CPS middlewares such as NASA's core Flight Executive (cFE) [5] and ROS [15]. Each component is encapsulated as a set of processes, as in the work of Shye [21] and Döbel [2]. Because CPSs interact with the environment, their components must adhere to strictly bounded execution times: S^3RES provides predictable timing for fault detection and recovery. S^3RES is meant to protect applications against uncorrelated single bit upsets in registers and main memory; it does not address faults in kernel space or in the voting mechanisms.

This paper makes the following contributions:

1. Introduce the design and implementation of S^3RES, a fault tolerant middleware infrastructure with predictable recovery.
2. Explore the use of the POSIX interface by such a system.
3. Evaluate the system performance through micro-benchmarks and fault tolerance through a simulated failure injection campaign.

2. RELATED WORK

Mukherjee, Kontz, and Reinhardt [13] use the redundant hardware cores of commodity Symmetric Multiprocessing (SMP) architectures to implement hybrid transient fault protection. Software threads were duplicated across multiple hardware threads using hardware queues and buffers to duplicate `load` instructions and vote on `store` instructions before committing. Later work demonstrated that small losses in fault tolerance could lead to greater performance gains [14], however Mukherjee et al. acknowledge that small hardware changes are difficult to make due to the complicated design of SMP architectures. Quest-V [12] and Maestro [24] target multi-core platforms with software-only mechanisms. Quest-V is a chip-level distributed system which uses a hypervisor to maintain redundant virtual machines (VM) and calculates hashes of VM memory to check for consistency. Maestro implements a form of process level redundancy on TILE64 processors that spawns replicas on file `open` operations and votes upon `close`. S^3RES is a software-only framework which is suitable for single and multi-core architectures.

The OSEK compliant embedded RTOS *d*OSEK [8] is a software-only approach that uses static techniques to reduce the amount of code vulnerable to transient faults: live kernel state is minimized, loops are unrolled to prevent control flow errors, and system calls are inlined to prevent return address problems. These instruction level techniques and others such as duplicating variables and checking control flow have also been applied to application source code [16]. These techniques may complement S^3RES well: process replication protects components, but instruction level mechanisms would provide a way to protect the code used to vote on replica outputs.

Microkernels have been shown to offer some degree of transient fault tolerance by virtue of their design alone: CuriOS [1] recovers from over 80% of arbitrary bit flips injected into system services, without the aid of voting mechanisms. Fiasco.OC [2] is a microkernel that achieves higher fault tolerance through software redundant multithreading using a minimized reliable computing base. Song and Parmer [22] have extended the Composite component-based OS to detect and recover from latent faults in system services by checking that messages produced by components conform to a system model. S^3RES only provides protection for user-space applications and does not attempt to protect system services. Pairing S^3RES with a fault tolerant OS may increase overall system reliability.

Shye, Blomstedt, Moseley, et al. [21] introduced PLR, which

S^3RES is available at https://github.com/jcmarsh/S3RES

EWiLi'16, October 6th, 2016, Pittsburgh, USA.

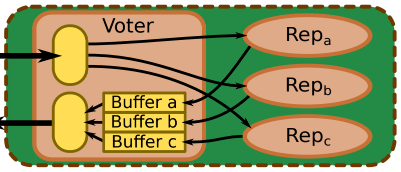

Figure 1: Protected component using TMR

replicates computation at a process level using system call emulation to run single threaded applications as three replica processes and a figurehead process. The authors show that 10% to 65% of SEUs (dependent on workload) do not impact the output of the replicas, and are thus benign. Their PLR implementation detects 100% of non-benign faults injected, and `fork` is proposed as a recovery mechanism. S^{3}RES does not use system call emulation; instead the input and output channels of software components are redirected to a user-space voter process. This matches the component based approach employed by CPSs and allows redundancy to be scaled based on the function of individual components. S^{3}RES is also able to maintain real-time guarantees and implements recovery using `fork()`, the implications of which we explore here.

3. DESIGN

The goal of S^{3}RES is to explore *scalable modular redundancy* in CPSs at a component level, within the POSIX specification and with real-time constraints. Components are a unit of software, typically a process, which runs continuously and communicates primarily through message passing. The faults considered are SEUs which result in execution errors (i.e. segmentation fault), control flow errors, silent data corruption (SDC), or are benign. Scaling redundancy for components allows system designers to make trade offs between task performance, resource requirements, and fault tolerance. We choose a component based approach because:

1. The concept of a component is pervasive in the CPS domain. Many middlewares provide for some form of component with message passing capabilities.
2. Coarse grained units of replication have been shown to be more efficient due to lower false positive rates [21].
3. Components are designed to be modular and interchangeable with well defined interfaces, providing a natural boundary to test for faults and to prevent error propagation. They are often the unit of fault isolation in systems, supported by hardware memory isolation mechanisms and threads for temporal isolation.

We model the architecture of middleware frameworks such as ROS and cFE, which construct CPSs as graphs of communicating components. Each component encapsulates some functionality, such as a path planner, and communicates through message passing. We refer to these units as *replicas*, which constitutes a sphere of replication [13] in S^{3}RES. To create a *protected* component, S^{3}RES creates multiple replica instances and a *voter* that maintains consistency between the replicas, detects faults, and facilitates recovery.

In order to interact with the physical world a CPS must stay responsive to changes in its environment: the components of the system must be able to respond predictably. A subset of these real-time components are reactive: they compose the most safety critical control loop of the system and are assumed to run once per control loop period. Reactive components may only produce one message for each outgoing pipe per control loop, simplifying the voter. This is intended to decrease voter surface area and ease hardening measures to be explored in future work. Real-time, non-reactive components and non real-time components may output multiple messages per pipe and may run at different rates than the reactive control loop.

3.1 The Voter

The voter coordinates a scalable number of replicas and arbitrates between them and the rest of the CPS. To the rest of the system, a protected component is identical to an unprotected one in all aspects except reliability and timing properties. Figure 1 shows a protected component with one incoming and one outgoing channel. The voter interposes upon the communication channels: incoming messages are copied and passed to each replica while outgoing messages from the replicas are buffered and compared for consistency by the voter. This allows the voter to detect SDC.

The voter tracks response time to detect control flow and executions errors. A control flow error (that does not become an SDC and is not benign) will result in the replica being halted for exceeding its timeout period. An execution fault triggers a hardware exception, which the voter detects indirectly by the replicas failure to respond.

The number of replicas dictates the capabilities of the voter. With one replica, the voter is limited to detecting execution and control flow errors while SDCs go undetected. Recovery losses component state, as no healthy replica is available. With two replicas, execution and control flow errors can be recovered from by replacing the failed replica with a copy of the remaining healthy replica. SDCs can be detected, but the voter will not be able to tell which replica the fault occurred in. With at least three replicas an SDC can be detected and recovered from by the voter.

The replica recovery process when no healthy replica exists is the same as initial startup. When a healthy replica is available, the voter coordinates with a healthy replica to create a replacement:

1. The Voter detects that a fault has occurred, identifies the faulty replica, R_f, and selects a healthy replica, R_h.
2. R_h creates a new copy of itself, R_n.
3. R_n establishes communication with the Voter.
4. The Voter cleans up R_f, updates its state related to R_n, and R_h.

3.2 POSIX

The choice of the widely supported POSIX interface gives our system a large potential user base and allows us to leverage POSIX defined functionality in existing OSes. The steps of the recovery process dictate in part the S^{3}RES design requirements: table 1 lists the POSIX interfaces which we rely on. Voters and replicas are processes, which use `fork / exec` for initial startup. These processes are scheduled using `SCHED_RR` except in the case of non real-time components, and the voter uses `sched` operations to control the execution of replicas by manipulating priority. Communication is done through `pipes`, which the voter monitors using `select` with a timeout to detect control flow and execution errors The voter detects SDC by comparing replica outputs.

Once a fault has been discovered, the voter sends `SIGKILL` to the faulty replica and a user defined signal to a remaining healthy replica. The healthy replica's signal handler uses `fork` to create a replacement replica from the healthy replica's state. Upon starting, a replica uses `mlockall` with the flag `MCL_FUTURE` followed by a page walk with fake writes to ensure that all pages have been copied over and are resident. In the case of SMR, no healthy replica is available so `exec` is used to restart the replica. The Voter will also use `waitpid` in the case of SMR to reap the dead process so that zombie processes do not accumulate.

To reestablish communication with the voter, a replica connects to it through a Unix Domain Socket (UDS) that the Voter has opened at a known location. The voter shares the file descriptors of new `pipes` with the newly created replica (which may not be related to the Voter). The replica uses `getpid` to retrieve its `pid`, which it sends to the voter through the same UDS. The `pid` is used by the voter to signal the replica during future recoveries.

S^{3}RES attempts to conform to the POSIX interface, but several exceptions were made. The interface lacks a way to pin processes or threads to a specific CPU core; S^{3}RES uses the Linux specific `sched_setaffinity` to prevent core migration from impacting timing. Linux uses the copy-on-write optimization on calls to `fork`,

Table 1: POSIX Compliance: * POSIX.1-2001, †POSIX.1-2001 and POSIX.1-2008, ‡Not POSIX compliant

Interface	Command	S^3RES Use and Comments
Process	exec†, fork*, getpid†, waitpid*	Initial component startup, replica recovery, and reap SMR replicas.
Memory	mlock*	Ensuring memory isolation between replicas after fork with MLOCKALL
Pipe	pipe*, read*, write*, close*	Data flow between and within components.
Sched	sched_setscheduler†, setpriority† sched_setaffinity‡	Used to set policy (SCHED_RR) and priority. Set core affinity.
Select	select*	Voter uses to multiplex pipes with a timeout.
Signal	sigaction*, kill*	Clean up failed replicas (SIGACTION) and to initiate recovery (User defined).
Socket	sendmsg*, bind*, accept†, listen*	Re-establishing Voter / Replica communication. Uses PF_UNIX.

which we circumvent by using /proc/self/maps (not specified by POSIX) to walk the process's pages, as discussed in Section 4.3. To track time in the voter, we decided to use architecture specific time stamp functions (Section 4.2). POSIX provides timers through timer_create, however on Linux the user must raise the priorities of ktimersoftd threads to avoid priority inversions between replicas and voters. The time stamp functions used add less than 20 lines of code, so this solution was deemed preferable to using nanosleep which would require additional processes to communicate with the voter.

4. IMPLEMENTATION

Here we detail both how S^3RES is implemented and the steps required for it to be used in a CPS. We assume that the CPS consists of executable programs that communicate through message passing via pipes. Each executable runs as a single process. S^3RES adds functionality to executables by building them with custom signal handlers and an initialization function to set up the signal handlers. The executable must call the added initialization function, and the signal handlers must have access to the replica's pipe information.

At system startup, a bootstrap program sets up the pipes that will connect unprotected and protected components together. Unprotected components are launched directly, with pipe information passed as command line arguments. The bootstrap program also launches the voter for each protected component which then launches the specified number of replicas, passed through command line arguments. The voter creates the replicas, duplicates pipes for each, and shares pipe file descriptors with the replicas via the same mechanism used during recovery (described in Section 4.4).

4.1 User Requirements

The bootstrap program requires the user to provide a description of the CPS. This description shall list the executable components that of the system, replication level, priority, and communication channels with associated timeout periods. The level of replication may be none, in which case communication channels do not have a timeout period as the component will be run without a voter. Otherwise, the specified voter process will be executed.

The bootstrap program itself accepts two file descriptors when launched: one for the input to the system and one for the output. In our test setup, a benchmarking process is used to translate between the simulation environment and the CPS that we have constructed, and this process creates the input / output pipes and launches the bootstrap program. In practice, hardware interaction would be done through unprotected components that generate input for and accept output from the rest of the system. S^3RES does not yet have any provisions to protect components that directly interact with hardware.

The system designer is responsible for the choice of OS and validating that the POSIX implementation chosen is sufficient. S^3RES only addresses faults at the application layer: an OS that is robust to transient faults may increase overall fault tolerance. The designer must validate that the OS's POSIX implementation is suitable for transient fault tolerance. Section 3.2 describes these issues as well as non-POSIX functionality that S^3RES requires.

The user must assign each component a priority, taking into account the control loops of their system and avoiding overlapping priority. An unprotected component runs at the specified priority, while a protected component runs the voter at the specified priority and replicas at lower priorities. The priorities should follow the order of execution of the safety critical control loop: the components dealing with input have the highest priority, with each successive link in the control loop having a lower priority. Care must be taken with communication between components to avoid priority inversion [19]: If a high priority component blocks while communicating with a low priority component, the system could fail if a medium priority component starves the lower priority component.

4.2 Fault Detection

The first step of the recovery process, discovering that a fault occurred, consists of two main tasks: detecting the fault and discovering the faulty replica. SDC is the most straightforward case: all outgoing messages from replicas are compared for consistency. If there is a discrepancy, the replica which sent the minority output is considered faulty. This requires three replicas, however faults can be detected with two replicas. Latent faults may occur if the output from a replica does not completely reflect its health.

To detect execution and control flow errors, the voter uses a timeout to detect when replicas fail to send output before their timeout period expires. We track time using rdtscll for x86 systems and with performance monitor unit commands for ARM, which are not POSIX compliant. For our tests we disabled clock scaling, although many architectures provide speed independent timers. Voters set the timeout for select() calls as the difference between the elapsed time since input received and the component's timeout period. The timeout period is related to pairs of pipes: an input pipe and output pipe for which each incoming message generates a response. These pairs are specified in the system startup script used for initialization. Input pipes do not have to be associated with an output, so a replica may execute without the voter being able to monitor it. In such cases the voter will not be able to detect a fault until the next input from a timed pipe. The voter has a higher priority than its replicas so it will be able to run even if a faulty replica is hogging the CPU. The component that generates the input for the faulty replica must also have a higher priority or it will be starved, resulting in deadlock. For timed input pipes that generate multiple output messages, the order of output messages should not be assumed: for cross core communication, the order in which pipes are written to may be different than the order in which they are available to be read from.

When a message arrives on an input pipe of a reactive component on a single core system the voter sends the message to replicas one at a time while setting the current replica to have the highest priority. The timeout is set per replica and if it expires, the currently running replica is determined to be faulty. For multi-core systems the timeout is for all replicas: the voter sends to all replicas at once and does not manipulate priorities of replicas on different cores. If the timeout expires, the voter determines the replica that has not yet responded to be the faulty replica. During recovery in single core systems the voter will have to ensure that all replicas have a chance

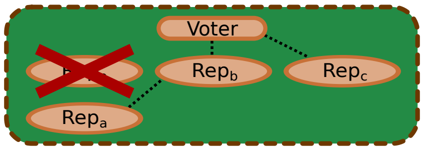

Figure 2: If a fault occurs in replica b, replica a will be orphaned.

to finish running (this is not an issue with multiple cores).

For non reactive and non real-time components, components with one to one relationships between input and output messages may send bursts of outputs. This is because the period of these components is not tied to the critical safety loop: multiple input messages may be buffered and then sent together, as the voter does not have knowledge of message size. To deal with this, the voter gives the highest priority to the replica that has output the least number of bytes on all pipes and will stop processing new inputs until all replicas are in sync. If a timeout occurs, the replica that is lagging behind the most is considered the faulty replica. This is a best effort measure for non real-time processes: replica priority (in our implementation, Linux niceness values) do not guarantee run order. Tracking CPU usage directly would be difficult, as the voter may not be related to the replicas (precluding the use of `rusage`, for example).

4.3 Replication

New replicas are created with `fork`, which copies almost the entire state of the replica into a brand new process, with exceptions such as signal masks and memory locks. So long as the copied process is healthy, the new process will be as well. However, many systems implement copy-on-write to make `fork` / `exec` calls more efficient. While copy-on-write is a known problem for real-time systems due to the possibility that it will result in unpredictable page faults, it invalidates the assumption that replicating a process can provide redundancy. For example, a process could `fork`, experience an SEU which modifies memory which is then read by the child process. The memory, which was not copied because neither process attempted to write to it, is now incorrect for both processes. To remedy this situation, when a new replica is created `mlockall` is called with `MCL_FUTURE`. S^{3}RES walks that process' memory using `/proc/self/maps`, making dummy writes to each page, ensuring that the page is copied over and is resident.

Leveraging `fork` to replace replicas has a troublesome consequence: replicas may become orphaned. This occurs when a replica, R_b, is forked to create another replica, R_a. R_a is now a child of R_b, so R_a will be orphaned if R_b experiences a fault. Figure 2 shows a component after it has experienced a fault. This leads to an uncommon use case for Linux: children routinely outlive their parent processes, for multiple generations. Anonymous memory pages, such as used for a process' stack and heap, are maintained in an `anon_vma_chain` structure, which grows by one with each generation. Upon the original parent's exit, in this case the voter, the structure is cleaned up. While the mainline fix was applied in January 2015, older kernels require a short patch [4], which prevents chains longer than five from being copied during a `fork`.

A final concern for orphaned processes: the voter will not be notified of their exit unless set as a sub-reaper with `prctl()`. This feature is only available in Linux for kernels 3.4 and newer and is not specified by POSIX. S^{3}RES does not depend on being notified of a replica's exit; the voter detects exits in the same manner as unresponsive replicas. Orphaned replicas are reaped by the `init` process, except for SMR protected component in which the voter must call `waitpid()` to prevent zombie processes from accumulating.

4.4 Communication

Once a new replica is created, it must establish communication with the voter. This is done through a Unix Domain Socket (UDS), which allows file descriptors to be shared between unrelated processes. The voter creates the UDS at a location known to the replicas during initialization. During recovery, the voter signals and raises the priority of R_h so that it is higher than the other replicas for that voter. The voter then blocks on an `accept` call, at which point R_h will receive the signal from the voter, call `fork` and return. The new replica, R_n, connects to the UDS to receive the file descriptors for the pipes. A string of meta data is also sent for each of the pipes; this data is not used by the voter, but is specified in the system description. Our example CPS uses this data to describe the type of messages sent over a pipe.

The pipes used for communication between the voter and replicas may have data buffered in them, but this data is not copied when a process is forked. This means that the voter must store a file descriptor for both the read and write ends of pipes that provide input to a replica. The voter must steal whatever is buffered by the pipes supplying input to R_h by reading them, and then write the data to both R_h and R_n once the new replica is forked. This ensures that both replicas have identical data.

The voter maintains a buffer of outputs from replicas, so this state must be copied over from this buffer for R_h when a new replica is created (for reactive components, the buffer only stores the last message sent). Normally these outputs are voted on and then written to the corresponding voter output pipe, but during recovery the data is assumed to be consistent under the assumption that the fault interarrival time is greater than the recovery time.

5. EVALUATION

Two sets of hardware were used to evaluate S^{3}RES; a BeagleBone Black ARM v7A locked at 1Ghz with 512MB of RAM and a computer with an Intel i7 quad-core processor with hyper-threading disabled, clock speed locked at 1.2Ghz, and 8GB of RAM. Three configurations of this hardware were used for benchmarks: 1Ghz ARM, 1.2Ghz x86_64 with a single core utilized, and 1.2Ghz x86_64 with all four cores utilized. Failure injection tests were performed on the ARM and x86_64 quad-core configurations. Both systems use the Linux kernel v4.4.3 with RT-Preempt patch rt9 configured to be fully preemptable. For micro-benchmarks, components were tested in isolation and either run with no voter (NMR), or with a voter and one, two or three replicas (SMR, DMR, and TMR respectively). To test the failure detection and recovery behavior, we implemented a component-based control system for a two-wheeled robot in a simulated 2d grid based maze using the Player/Stage robot simulator [6]. The simulator models the physics of the real world, in real-time, giving us a repeatable environment for testing actual control system code. The robot must navigate from one corner of the maze to the other without hitting any obstacles using its global location and sixteen distance sensors arranged in a uniform ring. The simulator was run on a separate computer on a local ethernet network.

5.1 Micro-Benchmarks

To evaluate the voter and measure the overheads it introduces, we ran a controller which echoes back incoming messages with a variety of voter configurations. Figure 3 shows the response times for the controller with four component configurations (NMR, SMR, DMR, and TMR) and the three hardware configurations. Each bar represents approximately 8000 round trip messages.

With each increase of redundancy level, the system has to send / receive one additional message. The size of the message is also varied from 8 bytes to 4096 bytes. The costs of message passing are high for the ARM system, and they appear to scale poorly as well. The quad-core x86_64 configuration performs less well than its single core counterpart for NMR, SMR and DMR setups, implying that the cross-core communication costs will outweigh the benefits of parallel processing for short running components.

Figure 4 shows the mean (with standard deviation) and observed WCET in microseconds for restarting components of various sizes.

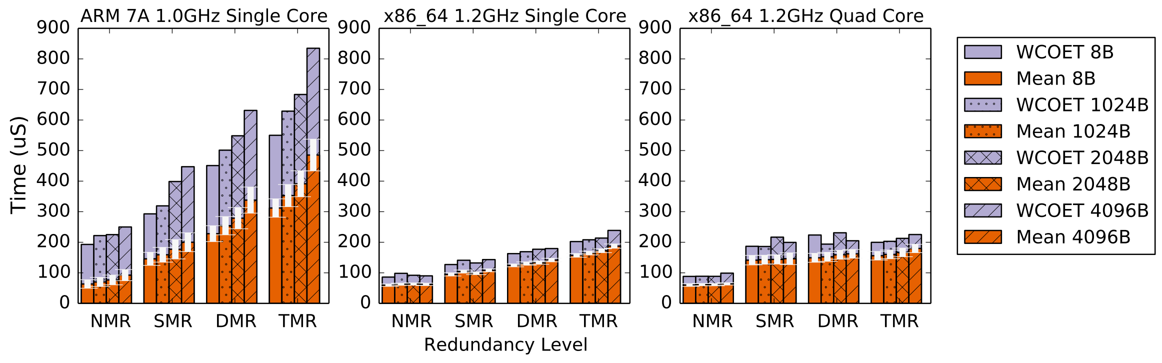

Figure 3: Time for a component to respond to a message compared against replication level and message size.

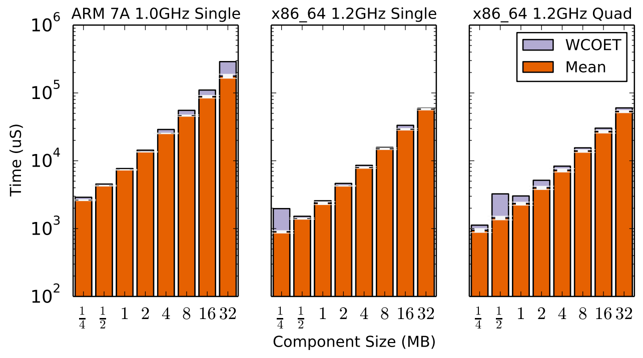

Figure 4: Replica restart timing vs. Component size. Error bars show standard deviation.

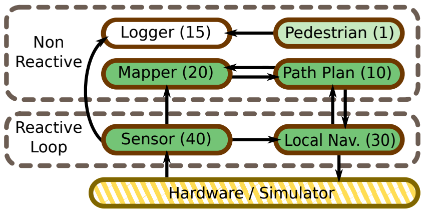

Figure 5: TMR core components, NMR logger, and SMR pedestrian detection. Base real-time priorities in parenthesis.

Each bar represents approximately 1500 restarts. The smallest size, 256Kb, is representative of the components used in our test setup. An empty component occupies approximately 216Kb when executed. In the four core configuration, the restarted replica was pinned to a different core than the replica it was created from. The results are similar across the single and quad-core configurations, suggesting that there is not a significant cost to migrating cores relative to overall restart costs. As one would expect, the 1GHz ARM processor takes longer to restart replicas.

5.2 Failure Injection

Figure 5 shows the components of the system used for the maze navigation task that we evaluated. The reactive components (sensor and local navigator) deal with sensor input and generate actuation commands. These components are responsible for detecting and avoiding obstacles, and are run at higher priorities than the rest of the system. The nominal rate for these components to run is 10Hz, triggered by input sensor data from the simulator. The next two components (mapper and A* path planner) map the environment and generate waypoints for the local navigator. These four components constitute the core system that we evaluate for fault tolerance by varying their level of redundancy from NMR to TMR together.

The final two components are used to measure the system and are exempt from failure injection. The logger component records the location and speed of the robot to a file. The pedestrian component runs the libccv [9] implementation of the Integral Channel Features pedestrian detection algorithm [3] repeatedly on a test image. This will serve as a computation bound background task for use in future work. Memory leaks in libccv forced us to run the component using SMR: the single replica calls `exit()` once finished and a modified voter is notified using `waitpid()` to start a new replica.

The priority of components is the same throughout all tests and configurations. The RT_Preempt patch allows for real-time priorities (from 1 to 99, lowest to highest) to guarantee execution order. A benchmarker component, which serves as the interface to the simulator as well as where we track system response times, runs at a priority of 98. The reactive layer components are the next highest: the sensor has a priority of 40 and the local navigator a priority of 30. The mapping component has a priority of 20, and the logging component a priority of 15. Finally, the path planner was run at a priority of 10. For protected components, the Voter runs with the stated priority, and replicas are run at a slightly lower priority. For example, the local navigator's voter has a priority of 30, and its three replicas have priorities of 28, 27, and 26. The pedestrian component is an exception: its voter has a priority of 1, and its single replica is run as a non real-time process with a niceness value of -11 (on a scale of 19 to -20, lowest to highest) to guide the scheduler. The components have priorities less than 50, because that is the priority at which interrupt handlers run at by default.

We simulate failures with signals to evaluate the fault detection and recovery characteristics of S^{3}RES. `SIGTERM` signals are used for execution errors, a signal handler that corrupts the next outgoing message for SDC, and a signal handler that causes to the replica to enter into an infinite loop for control flow errors. When injecting a failure, a signal is sent to a randomly selected replica. This does not simulate the relative exposure of each component, but is sufficient to evaluate how well the system recovers from faults. For the TMR configuration, all three types of failures were tested. Since DMR alone can not recover from SDC, only execution and control errors are injected for the DMR configuration. Failures were not injected into the SMR and NMR configurations: SMR loses component state during recovery causing the robot to become trapped in deadends of the maze and NMR crashes when any failure is injected.

Table 2 shows the error rate for the system in different configurations while performing the maze task. Injecting failures into the complete system as it performs a task is a more accurate measurement of the system's fault tolerance: injecting failures into isolated components masks communication dependent errors. Data was collected for fifty maze runs in each ARM configuration, and 100 for x86_64 Quad. The number of faults injected varied given task completion time, injection rate (2Hz for all except CFE injection on ARM), and task failures. The injection rate was based off the inter-arrival time of 500ms as calculated by Song [22] for which there is a very high probability for any place on earth that faults will not occur with a smaller inter-arrival latency. This rate had to be reduced to 0.2Hz for injecting control flow errors on the ARM system, which was unable to run reliably with the higher rate. The target pid of each injected failure was recorded, and then compared to the recov-

	ARM			x86_64 Quad		
Config	Injected	% Error	F Pos.	Injected	% Error	F Pos.
SMR No	0	NA	0	0	NA	0
DMR No	0	NA	0	0	NA	70
TMR No	0	NA	6	0	NA	48
DMR Exec	17248	0.2841	10	35171	0.0313	34
DMR CFE	1814	2.0397	28	29986	0.7637	453
TMR Exec	14380	0.4033	38	36599	0.0710	25
TMR CFE	1739	4.8879	57	24471	1.0870	239
TMR SDC	15070	0.5839	54	33034	0.5176	916

Table 2: Failures injected, percent of failures resulting in errors, and false positives for each configuration. Failure types: execution (Exec), control flow (CFE), and silent data corruption (SDC).

	ARM				x86_64 Quad			
Fault	NMR	SMR	DMR	TMR	NMR	SMR	DMR	TMR
None	100	100	100	100	100	100	100	100
Exec	-	-	96	84	-	-	93	96
CFE	-	-	39*	42*	-	-	81	54
SDC	-	-	-	99	-	-	-	81

Table 3: Successful runs out of 100 under different failure injection models and configurations for the ARM single core and x86_64 quad-core systems. *Only 50 control flow error injections were run with errors injected at a rate of 0.2Hz.

ery messages from the Voters to generate the error count. When a voter initiated the recovery of a replica that had not been selected by the failure injector, a false positive was counted. Control flow errors proved most difficult to recover from, especially on the ARM platform, and the quad-core injection results reveals issues with false positives. Even so, the failure rates for all configurations was below 5% with many significantly lower, showing potential for improved overall system reliability in the presence of SEUs.

Table 3 shows task outcome: the number of successful maze runs out of 100 runs with varied failures injected and replication levels. Each run lasts about 160 seconds and is considered successful only if the robot finishes the maze in under 200 seconds without hitting any walls. Failure injection was performed in the same manner as described above. The system performs fairly well when injecting execution faults and SDC, but control flow errors proved problematic for both ARM and x86_64 Quad. For ARM, only 50 CFE runs were performed for DMR and TMR: failed recovery often resulted in the system deadlocking making the tests difficult to automate. Even though recovery was not successful 100% of the time, each successful run recovered from approximately 320 failure injections.

6. CONCLUSIONS AND FUTURE WORK

This work presents a real-time application level framework for cyber-physical systems in which the redundancy of components may be scaled individually. We explore the use of POSIX features to implement the framework, such as using `fork` for recovery, and the difficulties in using the POSIX interface, such as ancestry issues, copy-on-write, and lack of CPU-pinning in the specification. Finally, we injected failures into an example system attempting to complete a maze navigation task to evaluate the fault tolerance of the system. The results show the potential of the approach: the highest failure rate measured was less than 5% and the system was able to complete the simulated task in the majority of runs, most of them with an 80% success rate despite failures being injected into replica processes at a rate of 2Hz.

In future work we intend to further explore the question of how to best scale redundancy for a CPS. Of interest are the trade-offs between task level performance, level of redundancy, and fault tolerance. This is a complicated problem as the fault tolerance of a system may be negatively impacted by fault protection mechanisms [18], is workload dependent, and depends on the tolerance of every part of the system. S^{3}RES provides a platform to explore these issues.

7. REFERENCES

[1] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Curios: improving reliability through operating system structure. In *OSDI*, 2008.

[2] B. Döbel, H. Härtig, and M. Engel. Operating system support for redundant multithreading. In *EMSOFT*, 2012.

[3] P. Dollár, Z. Tu, P. Perona, and S. Belongie. Integral channel features. In *BMVC*, 2009.

[4] D. Forrest and R. van Reil. Repeated fork() causes slab to grow without bound. https://lkml.org/lkml/2012/8/15/765, 2012. [Online; accessed 30-March-2015].

[5] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew. Verifying architectural design rules of the flight software product line. In *SPLC*, 2009.

[6] B. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *ICAR*, 2003.

[7] H. Heidt, J. Puig-Suari, A. Moore, S. h. Nakasuka, and R. Twiggs. Cubesat: A new generation of picosatellite for education and industry low-cost space experimentation. In *SmallSat*, 2000.

[8] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dosek: the design and implementation of a dependability-oriented static embedded kernel. In *RTAS*, 2015.

[9] L. Liu. Ccv-a modern computer vision library. http://libccv.org/. [Online; accessed 22-September-2015].

[10] W. Q. Lohmeyer, K. Cahoy, and S. Liu. Causal relationships between solar proton events and single event upsets for communication satellites. In *AeroConf*, 2013.

[11] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[12] E. Missimer, R. West, and Y. Li. Distributed real-time fault tolerance on a virtualized multi-core system. *OSPERT*, 2014.

[13] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *ISCA*, 2002.

[14] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. Slick: Slice-based locality exploitation for efficient redundant multithreading. In *ASPLOS*, 2006.

[15] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, 2009.

[16] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *CGO*, 2005.

[17] R. Ridenoure, R. Munakata, A. Diaz, S. . Wong, B. Plante, D. Stetson, D. Spencer, and J. Foley. Lightsail program status: one down, one to go. In *SmallSat*, 2015.

[18] H. Schirmeier, C. Borchert, and O. Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *DSN*, 2015.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *TC*, 1990.

[20] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, 2002.

[21] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *TDSC*, 2009.

[22] J. Song and G. Parmer. C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *RTAS*, 2015.

[23] M. A. Swartwout. Cubesat database. https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database. [Online; accessed 7-April-2016].

[24] J. P. Walters, R. Kost, K. Singh, J. Suh, and S. P. Crago. Software-based fault tolerance for the maestro many-core processor. In *AeroConf*, 2011.