Scaled Redundancy in Cyber-Physical Systems

James Marshall *[1], Gedare Bloom[2], Gabriel Parmer[1] and Rahul Simha[1]

[1]Department of Computer Science, The George Washington University
[2]Department of Computer Science, Howard University

Abstract

Cyber-physical systems (CPSs) have stringent requirements for power, size, cost and reaction time. Fault protection mechanisms negatively impact 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 demonstrate a %10 speedup of task performance with maze navigation and pedestrian detection tasks by scaling redundancy, with no loss of fault tolerance.

I. Introduction

Transient faults in cyber-physical systems (CPS) such as satellites may be caused by electrical noise or radiation [1]. These single event upsets (SEUs) manifest as "bit-flips" and may cause data corruption, computational errors, and execution faults. A common protection mechanism is to provide double and triple modular redundancy (DMR and TMR) through redundant hardware [2], but this increases power and volume requirements. Radiation hardened processors are an alternative, but lag behind commodity hardware in performance and cost.

Software solutions are able to provide near equivalent protection by replicating computation *temporally* instead of *spatially*. The additional computational load is offset by using faster and cheaper commodity parts. Software solutions allow the granularity of computational units to be scaled. Coarser-grained units, such as a process, reduce the cost of reliability by lowering voting overheads and allowing benign SEUs to be ignored [3]. We posit that for CPSs, scaling the level of redundancy of certain units can be achieved without sacrificing reliability because CPSs comprise robust subsystems that are resilient to silent data corruption (SDC), further reducing costs.

We present Scalable System Support for Reliable Embedded Software (S^{3}RES) to enable options in scaling redundancy to reduce the cost of reliability. S^{3}RES is a software-only, user-space approach to SEU detection and recovery that interposes between a control application and a POSIX compliant real-time operating system for uniprocessor systems. Redundancy may be scaled from TMR, DMR, to no redundancy on a *per-component* basis. Components are a common abstraction defined by CPS middlewares such as NASA's core Flight Executive (cFE) [4] and ROS [5]. Without protection, components are vulnerable to SUEs, which are categorized by their effects as follows:

1. Benign: The SEU causes no observable change. *A register is altered prior to a write.*
2. Control Flow Error: Causes a branch in process flow, altering execution time. *A change to the program counter, a jump address, or a branch's condition.*
3. Execution Error: Causes the process to trip OS protection mechanisms. *A memory address is altered to an out of range value and then accessed.*
4. Data Corruption: The functional output of a process is changed. *A register is altered prior to a read.*

Because CPSs interact with the environment, their components must adhere to strict bounded execution times. S^{3}RES is able to maintain real-time guarantees while being subjected to a fault injection campaign as demonstrated in previous work [6].

This paper describes our preliminary results showing that scaling redundancy can improve task performance and explains our plan to explain the relationship between scaled redundancy, fault protection, and task performance.

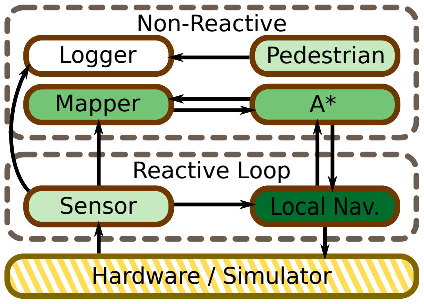
II. System Design

Our goal with S^{3}RES is to explore the application of *scalable modular redundancy* to CPSs at a component level. To do this, we model the architecture of middleware frameworks such as ROS and cFS, which construct systems for CPSs as graphs of communicating nodes. Each node encapsulates some functionality, such as a path planner, and communicates through message passing.

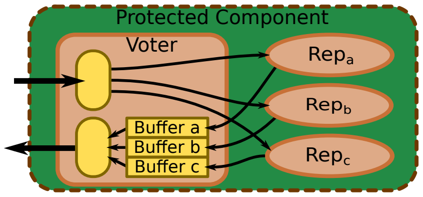
Figure 1a shows the components of an example robot control system. The interface with our simulated hardware, a two wheel robot navigating a 2d maze in Player/Stage [7], is shown at the bottom. Directly above that are the sensor and local navigator components, which detect and avoid obstacles. The next two components—the mapper and A* path planner—map the environment and generate waypoints for the local navigator. These four components constitute the core system.

The final two components are used to measure the system: the logger component records the location and speed of the robot to a file and the pedestrian component runs an implementation of the Integral Channel Features pedestrian detection algorithm repeatedly on a test image. This provides a realistic example for a computation intensive task and allows us to measure the impact of redundancy on throughput. The pedestrian component has memory leaks,

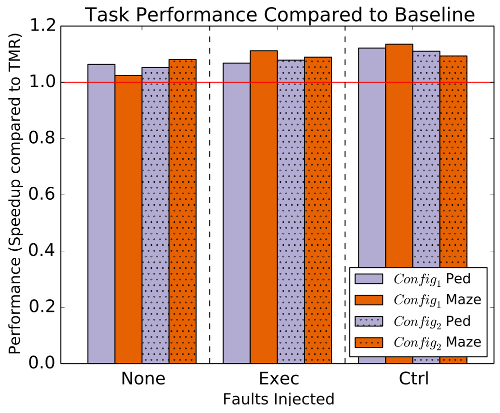
*jcmarsh@gwmail.gwu.edu



(a) Configuration 2: SMR for Sensor, DMR for Mapper and Path Planner, TMR for Local Navigator.



(b) Component protected with TMR



(c) Task speedup compared to the baseline configuration. Seconds per pedestrian detected in violet, maze completion time in orange.

Figure 1

so it is periodically restarted using a single replica with a voter, refered to as single modular redundancy (SMR).

To create a protected component, the original component is replaced with a voter and a scalable number of replicas of the original component. Figure 1b shows an example protected component with one incoming and one outgoing channel. The voter interposes upon the communication channels: incoming messages are duplicated and passed to each replica while outgoing messages are buffered in the voter and then compared for consistency. The voter detects control flow errors and execution faults by setting a timeout for replica responses.

The level of replication dictates the capabilities of the voter. With SMR, the voter is able to detect faults and control flow errors. Recovery will lose component state, as no running replica will be available, and SDCs will go undetected. With DMR, faults 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 by comparing outputs, but the voter will not be able to tell which replica the fault occurred in. It is only with TMR that a SDC can be detected and fully recovered from.

III. Task Performance

To evaluate S^{3}RES, we injected faults into three configurations of the system shown in Figure 1a while the robot navigated through a maze using its global location and sixteen distance sensors. The baseline configuration uses TMR for all four core components, and $Config_1$ reduces the mapper and path planner to DMR. $Config_2$ does as well, and reduces the filter to SMR.

The system is able to perform well when faults are injected into SMR protected components that maintain no vital internal state, such as filter. We need to expand S^{3}RES to be able to evalute performance with SDC errors. For components such as filter, SDC may not impact performance if all dependent components are robust to noisy inputs. For components such as the path planner, we need to implement a recovery mechanism such as checkpointing, for which DMR will provide fail-stop behavior.

Figure 1c shows speedup of $Config_1$ and $Config_2$ compared to the baseline configuration. The groupings, from left to right, show performance with no faults, execution faults, and control faults. The speedup observed is for task based performance: by freeing resources, the robot is able to complete the maze faster while detecting more pedestrians. With faults injected, the scaled configurations experience less of a degradation to performance while still being as resilient to faults as the baseline configuration.

IV. Conclusions and Future Work

With S^{3}RES established as a suitable platform to explore scaling redundancy, we intend to expand upon our preliminary results regarding task performance. We intend to further investigate trade-offs between task performance and level of redundancy, as well as how to characterize components to help determine the level of redundancy to use, and whether or not complimentary forms of protection are applicable.

References

[1] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398. IEEE, 2002.

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

[3] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *DSN*, 6(2):135–148, 2009.

[4] Dharmalingam Ganesan, Mikael Lindvall, Chris Ackermann, David McComas, and Maureen Bartholomew. Verifying architectural design rules of the flight software product line. In *SPLC*, pages 161–170, 2009.

[5] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[6] James Marshall, Gedare Bloom, Gabriel Parmer, and Rahul Simha. n-modular redundant real-time middleware: Design and implementation. *submitted for publication*.

[7] Brian Gerkey, Richard T Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *ICAR*, volume 1, pages 317–323. IEEE, 2003.