Automated Software Compiler Techniques to Provide Fault Tolerance for Real-Time Operating Systems

Benjamin James
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA
b_james@byu.edu

Jeffrey Goeders
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA
jgoeders@byu.edu

***Abstract*—In this work we explore applying automated software fault-tolerance techniques to protect a Real-Time Operating System (RTOS) and present experimental results showing that these programs can achieve anywhere from 1.3x–257x improvement in MWTF.**

I. Introduction

One approach to protect software in radiation-prone environments is to modify the software code itself to be tolerant of single event upsets (SEUs) [1]–[4]. Previous works have shown techniques including modifying the assembly code by hand [5], [6], using automated tools to instrument the code, or reliance on specific architectural features [1], [7]. In recent years, we published a tool titled COAST (**Co**mpiler-**A**ssisted **S**oftware Fault **T**olerance) [8], [9], which takes a different approach, and applies protection to the intermediate representation (IR) of the code during LLVM compilation. COAST has been evaluated in radiation tests that show mean work to failure (MWTF) improvements of 1.2x to 36x [9].

One major limitation of our past work with COAST was that we only targeted simple bare-metal applications, such as matrix multiplication or AES encryption. The objective of this work is to explore whether automated COAST protection could be applied to an entire operating system, or if the complexities of an operating system would make this infeasible. We apply automated fault-mitigation to the popular Real-Time Operating System (RTOS) FreeRTOS. Adding COAST protection to this RTOS required enhancements to the open-source COAST tool and giving the tool specific configuration options to handle the complexities of FreeRTOS.

Specifically, some of the challenges we encountered related to maintaining automated protection while switching between multiple threads of execution, accessing kernel-specific objects, and the fact that part of the kernel is written in assembly code. Our work demonstrates that automated protection tools can be used on complex systems, with some assistance from the user. The key contributions of this work are:

- A demonstration of compiler-automated fault mitigation for a full real-time kernel (FreeRTOS).
- Fault injection results demonstrating 1.3x–257x improvement to MWTF (Mean Work To Failure)
- A set of open-source tools and configurations used in the work, available at https://github.com/byuccl/coast.

Benjamin James and Jeffrey Goeders are also members of the NSF Center for Space, High-Performance, and Resilient Computing (SHREC)

II. Related Work

Past work has investigated protecting RTOSes; however, most of these works manually protect the scheduler code, rather than applying compiler-automated protection to the entire kernel. Different techniques include checkpointing [10], building in lots of slack to the rate-monotonic scheduling [11], dynamic scheduling [12] and others [13]–[16]. In addition, SafeRTOS is an RTOS marketed for safety critical applications.

The methods above focus on protecting the control-flow of the RTOS, whereas COAST attempts to protect the entire data flow of the program, and provides synchronization and correction on each control-flow branch. These methods work on a coarse-grained scale, compared to COAST's voting code sprinkled throughout both application and kernel code. These methods are different than, though possibly complementary to, the methods discussed in this work. Some more closely related work has been done by Borchert et al., who use Aspect Oriented programming to get automated protection of two different RTOS platforms [17], [18]. This depends on the Aspect-Oriented C++ compiler, so RTOSes written in C, such as FreeRTOS, could not be supported.

III. Protecting a Real-Time Operating System

We use COAST to protect the entire software stack, both application and kernel code. Our intent was to use COAST out-of-the-box to protect all of the user variables and kernel objects with triple redundancy (TMR); however, while attempting to implement this protection approach we ran into several issues. This section describes these challenges and solutions.

First, some of the code for the kernel is written in assembly code. This is not possible to protect with COAST, as architecture-specific assembly code is not part of the LLVM IR. Second, issues of object duplication arose from the various ways that FreeRTOS allocates objects, and originally we were losing copies of kernel objects. We also had to deal with ensuring that the correct number of copies of dynamically allocated objects are created, as well as making sure function inlining did not interfere with the Scope of Replication.

Even with all of these difficulties, we were still able to implement the protection schemes solely during compilation

978-3-9819263-5-4/DATE21/©2021 EDAA

using the COAST tool. COAST required a large number of command line flags and in-code directives to properly maintain the integrity of the Scope of Replication, but we did not hand-modify any assembly code for this experiment.

A. Architecture Specific Code

As the COAST tool operates on LLVM IR, it is dependent on having the source code available. Anything already at a lower level than the IR (such as assembly code) cannot be modified by an LLVM-based tool. Thus, any global variables accessed by the assembly must be excluded from the *Scope of Replication* (SoR). The SoR defines which parts of a program (ie., which functions and variables) should be replicated. Protected variables, which are those within the SoR, should *only* be accessed within the SoR, to ensure that replicas are kept up to date and synchronized. Unprotected variables can be accessed by functions within the SoR or not. In many cases it is possible to detect these uses by examining the list of functions and globals that are protected and comparing that with actual uses of each of the globals. We added a verification step to the COAST tool to automatically detect when this rule is violated.

In the FreeRTOS kernel, the context switcher is written in assembly. To handle this, we manually created a list of global variables which needed to be excluded from protection in order for the integrity of the SoR to be maintained, and instructed COAST to skip replicating these variables.

B. Losing Replicated Objects

Certain kernel functions create objects to be used by the user-space tasks. If the handles to these objects are returned directly to the caller via the function return value, copies can be lost. We have encountered a variant of this problem before; any function that dynamically allocates memory and returns a pointer to that value can lose the replicated copies on the function return.

One way to deal with this problem is by calling the function more than once, with each invocation returning one of the triplicated object copies. While this works in most cases, certain functions have side effects, and calling them multiple times can cause internal data corruption. These functions must be invoked only once, and the function signatures must be modified to "return" multiple values. We added configuration options to COAST to allow users to specify which functions need to have return values triplicated. COAST can modify the function signature to add two extra arguments which are used to return the extra values through pointers. The call sites are also updated automatically to retrieve these triplicated objects.

C. Handling Dynamic Memory Allocation

Functions which dynamically allocate memory are some of the most important function calls to be aware of when replicating and protecting the data of a program. In some instances it is appropriate to call these functions multiple times, and other instances they must only be called once.

As an example from FreeRTOS, allocating stack space for a task should only be done once. This is because the stack is something which is primarily used by assembly code; LLVM IR has no concept of a system stack. In contract, when allocating the *Task Control Block* (TCB), a struct which contains data for each user task, we want the data to be replicated, and thus multiple memory blocks should be allocated. Unfortunately there is no obvious way to do this automatically, and again we needed to manually build a list of locations in the code where dynamic allocation should not be replicated.

D. Inlining

Compilers often inline the bodies of functions into their callers as an optimization. In most cases this is desirable, as it can lead to decreased run-time. However, if one of the functions is supposed to be inside the SoR, and the other is not, inlining can corrupt the protection scheme. When a function is inlined, the symbol for the original function no longer exists, and so COAST cannot treat it correctly because it is no longer distinct from its caller(s). There were a few functions in the FreeRTOS kernel where we had to instruct the compiler not to inline.

E. Partial Protection Schemes

In our past work we have noted that there is naturally a significant penalty in both run-time and memory usage for comprehensive TMR protection ($\sim$3-4x increase). Thus, in addition to our protection scheme which covers all of the kernel and application code, we also wanted to explore the effects of protecting only the application code in order to better understand the possible trade-off between fault tolerance and performance cost. The main consideration for this protection scheme is that handles to the kernel objects cannot be replicated in the application code. This includes function arguments that serve as return values.

We also considered protecting only parts of the kernel, and performed profiling to determine which parts of the kernel were invoked most frequently. However, ultimately we found that kernel functions and kernel objects were simply too interconnected to implement such a partial protection scheme.

IV. Experiment

A. Fault Injection Methodology

To determine the fault tolerance we custom created an automated fault injection setup. Our fault injection framework uses the QEMU (Quick EMUlator) tool to emulate the processor under study, the ARM Cortex-A9. QEMU has a GDB interface that we use to modify values in the guest program. We use a Python supervisor script which runs the application, each time changing a value in memory or system registers, and parses the output of the application to see if any errors occurred.

Our fault injection targeted three areas: 1) all processor caches, 2) only the data cache, and 3) the processor registers. We target the caches specifically because the A9 platform we have been testing (Xilinx PYNQ board) has unprotected SRAM caches. As DRAM (main memory) is about an order of magnitude less susceptible to failures than SRAM [19], we chose to specifically focus on cache faults. For each execution of a benchmark, a single fault is injected at a random cycle number since execution began, and in a random bit of the cache or register file.

TABLE I: Fault Injection results

Benchmark	Section	Runs	Faults	Errors	Timeout	Invalid	Error rate		MWTF	Size (KB)	Cycle count (avg)
rtos_kUser (Unmitigated)	cache	5000	0	122	97	0	2.44%		-	824	4.22×10^7
	dcache	5000	0	162	242	0	3.24%		-		
	registers	5000	0	150	261	0	3.00%		-		
.TMR (kernel + application)	cache	10000	292	44	359	0	0.44%	(**5.55x** ↓)	**1.495x**,↑	668 (0.81x)	1.56×10^8 (3.71x↑)
	dcache	10000	753	69	675	7	0.69%	(**4.70x** ↓)	**1.266x**,↑		
	registers	10000	323	47	450	0	0.47%	(**6.38x** ↓)	**1.721x** ↑		
.app.TMR (app only)	cache	10000	40	45	237	0	0.45%	(**5.42x** ↓)	**3.310x** ↑	592 (0.72x)	6.90×10^7 (1.64x↑)
	dcache	5000	49	45	234	2	0.90%	(**3.60x** ↓)	**2.198x** ↑		
	registers	5000	9	38	288	1	0.76%	(**3.95x** ↓)	**2.410x** ↑		
rtos_mm (Unmitigated)	cache	5000	0	60	181	0	1.20%		-	808	1.33×10^8
	dcache	5000	0	528	1145	4	10.56%		-		
	registers	5000	0	39	185	0	0.78%		-		
.TMR (kernel + application)	cache	60000	5962	10	477	2	0.02%	(**72.00x** ↓)	**20.767x** ↑	616 (0.76x)	4.61×10^8 (3.47x↑)
	dcache	60000	44549	11	333	1	0.02%	(**576.00x** ↓)	**166.135x** ↑		
	registers	10000	39	102	428	0	1.02%	(**1.31x** ↑)	**4.534x** ↓		
.app.TMR (app only)	cache	135000	13147	20	838	3	0.01%	(**81.00x** ↓)	**23.670x** ↑	563 (0.70x)	4.55×10^8 (3.42x ↑)
	dcache	150000	115724	18	778	2	0.01%	(**880.00x** ↓)	**257.157x** ↑		
	registers	5000	10	75	198	0	1.50%	(**1.92x** ↑)	**6.581x** ↓		

In our testing, we used two different benchmarks. The first is a modified version of the demo application distributed with FreeRTOS, which we call `rtos_kUser`. This application tests the use of many different kernel functions. The second, `rtos_mm`, runs a matrix multiplication alternating between two tasks. By using these two applications, we aimed to show the difference between an application dominated by kernel calls, versus one dominated by user code.

For each of these applications, we implement the protection scheme that covers as much of the code as possible, denoted with the ".TMR" suffix, which indicates that the benchmark (application and and kernel code) will be protected by the TMR protection. We also implement a protection scheme with a smaller overhead, which only protects the application part of the code. This version of the benchmark is denoted ".app.TMR".

B. Results

The injection results are shown in Table I, and a summary of MWTF improvements in Figure 1. In the results, an "Error" is when the the calculated result is incorrect. A "Fault" means that a fault was corrected by the COAST-inserted code. An "invalid" result means the benchmark output did not match the expected format, and "timeout" is when the benchmark did not complete in the expected time. We are most interested in reducing the error rate, as this represents silent data corruption (SDC), and would normally be undetectable. The next set of columns provide the error rate and mean work to failure (MWTF). The error rate is the number of errors divided by the total number of times the benchmark was run. While error rate is a good indication of our effectiveness at reducing the number of errors, it is important to recognize that increases in program run-time would naturally cause more faults to occur in an SEU prone environment. Thus, the MWTF column scales down the improvement in error rate by the increase in run-time, to provide a more fair metric of total improvement.

The final set of columns give the executable size and number of simulator cycles to complete execution. The reader may notice that the "size" column in each table indicates that the protected version of the benchmarks is actually *smaller* than the original size. This is because all unused symbols were stripped out during compilation, a feature of COAST.

`rtos_kUser`: Both protection schemes provided a decrease in error rate and related increase in MWTF for all three target sections. Note that even though total code protection resulted in an overall lower error rate, because of the steep penalty in run-time overhead it has, protecting only the application code gave a better increase in MWTF. This indicates there are some instances where protecting only the application code could provide enough fault tolerance to meet the desired goals, especially when reducing protection overhead is important.

`rtos_mm`: For this benchmark, while COAST definitely helps protect against errors in the caches, errors in registers resulted in a worse error rate than without protection. Errors caused by injections into registers are usually control flow errors. Since this benchmark is a triply-nested loop, any error in control flow is likely to mess up the hash that is run at the end to validate the result.

It is important to keep in mind that the number of bits in the register file is significantly smaller than the total bits in all the caches. This means that, in an environment where upsets are evenly distributed between all bits in the system, the results for injecting into the cache will most closely resemble the actual fault coverage.

C. Impact on Run-time and Memory Usage

To measure run-time, we performed detailed profiling of each application. As expected, run-time increased by ~3–4x, due to triplicated instructions and inserted voters. As evident from the *total* bars in Figure 2, `rtos_kUser` spends most of time in kernel code, while the `rtos_mm` benchmark is the reverse.

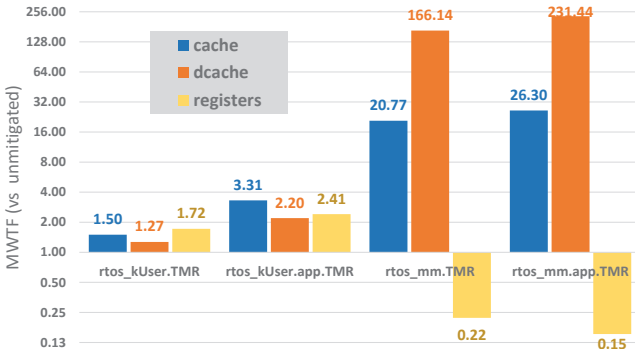

Fig. 1: MWTF Summary

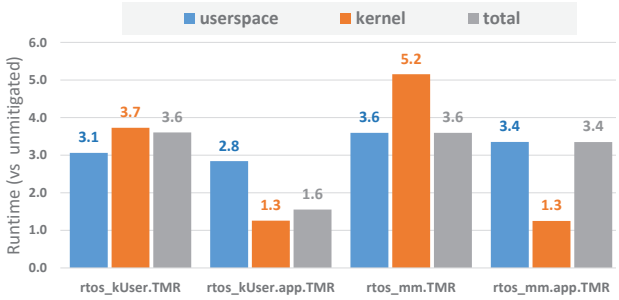

Fig. 2: Benchmark Run-time Overhead

To evaluate impact on memory usage, we measured both static and dynamic memory allocation. Static memory usage was determined by examining the ELF symbol table. Dynamic memory usage was calculated by manually searching for all memory allocation calls and using our knowledge of the applications to calculate how much memory these calls will account for. Total memory usage of each benchmark configuration, relative to the unmitigated version, is shown in Figure 3.

Compared to the change in run-time, the difference in memory usage is not as large. Some of this can be attributed to constant space used by the printing libraries and other similar functions. Another contributing factor is that there are certain parts of the kernel, most notably the task stack spaces, that cannot be replicated in any case.

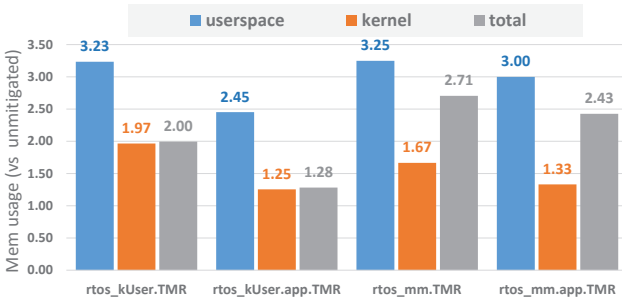

Fig. 3: Benchmark Memory Usage Overhead

References

[1] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar, M. A. G. Silveira, L. Ost, R. Reis, S. Cuenca-Asensi, and F. L. Kastensmidt, "Reliability on ARM processors against soft errors through SIHFT techniques," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2208–2216, Aug. 2016.

[2] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, F. L. Kastensmidt, R. Reis, and S. Cuenca-Asensi, "Reliability on ARM processors against soft errors by a purely software approach," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Sep. 2015, pp. 1–5.

[3] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective software-only error-detection technique using assertions," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3088–3095, Dec. 2015.

[4] E. Chielle, F. L. Kastensmidt, and S. Cuenca-Asensi, "Overhead reduction in data-flow software-based fault tolerance techniques," in *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design*, F. Kastensmidt and P. Rech, Eds., Cham: Springer International Publishing, 2016, pp. 279–291.

[5] 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, Mar. 2002.

[6] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Software resilience and the effectiveness of software mitigation in microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2532–2538, Dec. 2015.

[7] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *Proceedings of the International Conference on Dependable Systems and Networks*, IEEE, Jun. 2007, pp. 544–553.

[8] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Microcontroller compiler-assisted software fault tolerance," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 223–232, Jan. 2019.

[9] B. James, H. Quinn, M. Wirthlin, and J. Goeders, "Applying compiler-automated software fault tolerance to multiple processor platforms," *IEEE Transactions on Nuclear Science*, vol. 67, no. 1, pp. 321–327, Jan. 2020.

[10] L. Xu, Y. Bai, K. Cheng, L. Ge, D. Nie, L. Zhang, and W. Liu, "Towards fault-tolerant real-time scheduling in the seL4 microkernel," in *International Conference on High Performance Computing and Communications*, Dec. 2016, pp. 711–718.

[11] S. Ghosh, R. Melhem, D. Mossé, and J. S. Sarma, "Fault-tolerant rate-monotonic scheduling," *Real-Time Systems*, vol. 15, no. 2, pp. 149–181, Sep. 1, 1998.

[12] P. Mejia-Alvarez and D. Mosse, "A responsiveness approach for scheduling fault recovery in real-time systems," in *Real-Time Technology and Applications Symposium*, ISSN: 1080-1812, Jun. 1999, pp. 4–13.

[13] K. Kim, "ROAFTS: A middleware architecture for real-time object-oriented adaptive fault tolerance support," in *International High-Assurance Systems Engineering Symposium*, Nov. 1998, pp. 50–57.

[14] H. Kim, S. Lee, and B.-S. Jeong, "An improved feasible shortest path real-time fault-tolerant scheduling algorithm," in *International Conference on Real-Time Computing Systems and Applications*, ISSN: 1530-1427, Dec. 2000, pp. 363–367.

[15] H. Chen, W. Wang, W. Luo, and J. Xiang, "A novel real-time fault-tolerant scheduling algorithm based on distributed control systems," in *International Conference on Computer Science and Service System (CSSS)*, Jun. 2011, pp. 80–83.

[16] C. Buckl, M. Regensburger, A. Knoll, and G. Schrott, "Models for automatic generation of safety-critical real-time systems," in *International Conference on Availability, Reliability and Security (ARES)*, Apr. 2007, pp. 580–587.

[17] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generic soft-error detection and correction for concurrent data structures," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 22–36, Jan. 2017.

[18] C. Borchert, "Aspect-Oriented Technology for Dependable Operating Systems," Dissertation, der Technischen Universität Dortmund, Dortmund, 2017, 244 pp.

[19] T. Semiconductors. (Jan. 5, 2004). "Soft errors in electronic memory – a white paper," [Online]. Available: http://www.tezzaron.com/media/soft_errors_1_1_secure.pdf (visited on 12/02/2020).