Program Conversion for Detecting Data Races in Concurrent Interrupt Handlers*

Byoung-Kwi Lee[1], Mun-Hye Kang[1], Kyoung Choon Park[2], Jin Seob Yi[3], Sang Woo Yang[3], and Yong-Kee Jun[1,**]

[1] Department of Informatics, Gyeongsang National University,
Jinju 660-701, The Republic of Korea
{lbk1116,kmh,jun}@gnu.ac.kr

[2] Aero Master Corporation, 668-1 Bangji-ri, Sanam-myeon,
Sacheon-si, Gyeongsangnam-do, Korea
gilsion@amc21.co.kr

[3] Korea Aerospace Industriers, LTD., 802 Yucheon-ri, Sanam-myeon,
Sacheon-si, Gyeongsangnam-do, Korea
{avionics,sangyang}@koreaaero.com

Abstract. Data races are one of the most notorious concurrency bugs in explicitly shared-memory programs including concurrent interrupt handlers, because these bugs are hard to reproduce and lead to unintended nondeterministic executions of the program. The previous tool for detecting races in concurrent interrupt handlers converts each original handler into a corresponding thread to use existing techniques that detect races in multi-threaded programs. Unfortunately, this tool reports too many false positives, because it uses a static technique for detecting races. This paper presents a program conversion tool that translates the program to be debugged into a semantically equivalent multi-threaded program considering real-time scheduling policies and interrupt priorities of processor. And then, we detect races in the converted programs using a dynamic tool which detects races in multi-threaded programs. To evaluate this tool, we used two flight control programs for unmanned aerial vehicle. The previous approach reported two and three false positives in these programs, respectively, while our approach did not report any false positive.

Keywords: Races, Concurrent Interrupt Handler, Threads, Embedded Software, Dynamic Analysis.

* This research was performed as a part of R&D program Air-BEST (Airborne Embedded System and Technology) funded by MKE (Ministry of Knowledge and Economy). This research was financially supported by the Ministry of Education, Science Technology (MEST) and National Research Foundation of Korea (NRF) through the Human Resource Training Project for Regional Innovation.

** Corresponding author: In Gyeongsang National University, he is also involved in the Research Institute of Computer and Information Communication (RICIC) and GNU Engineering Research Institute (ERI).

1 Introduction

Data races [10] are one of the most notorious concurrency bugs in explicitly shared-memory concurrent programs, because these bugs are hard to reproduce and lead to unintended non-deterministic executions of the program. A data race is a pair of concurrent accesses to a shared variable which include at least one write access without appropriate synchronization. Since these races lead to unintended non-deterministic executions of the program, it is important to detect the races for effective debugging. These races may also occur in embedded software which often includes concurrent interrupt handlers. A previous program conversion tool [14] for detecting data races in concurrent interrupt handlers converts the program into a semantically equivalent multi-threaded program so that an existing thread verification tool can be used to find the races in the original program using static race detection. Unfortunately, this technique reports too many false positives without any false negative, because it uses a static technique.

This paper presents a novel conversion tool that converts an original program with concurrent interrupt handlers into a semantically equivalent POSIX threads [3] considering real-time scheduling policies and interrupt priorities of processor. And then, we detect races in the converted programs using a dynamic detection tool [1,6], called Helgrind+, developed for multi-threaded programs. Helgrind+ is an extension of the Helgrind tool which is a Valgrind tool [17] to detect synchronization errors in C, C++, and Fortran programs that use the POSIX threads. We empirically compared our approach with the previous one using two flight control programs of unmanned aerial vehicle (UAV). The previous approach reports two and three false positives in each program, respectively, while our approach does not report any false positive with still more false negatives than the previous static approach.

Section 2 introduces concurrent interrupt handlers and explains previous techniques for detecting races in multi-threaded programs. Section 3 presents our conversion tool that converts each concurrent interrupt handler into a semantically equivalent POSIX thread. Section 4 empirically shows our approach is practical with two flight control programs of UAV. The final section concludes our argument.

2 Background

An interrupt [14] is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves part or all of its context and jumps to a special subroutine called an *interrupt handler*. Microprocessors allow interrupts to be ignored or recognized through the use of two special machine instructions: *disable interrupt, or enable interrupt*, respectively. In a real-time environment, every period of interrupt disabling should be as shortly as possible. Interrupt disabling may affect interrupt processing to be delayed or ignored, and then cause such interrupts to be missed. Most processors allow interrupts to be nested.

The structure of common embedded software has two major components as shown in Figure 1: concurrent interrupt handlers called *foreground routines* [7], and the main routine, called the background routine, which is one infinite loop. These concurrent interrupt handlers may also involve data races. Data races [10] occur when two parallel threads access a shared memory location without proper inter-thread coordination, and at least one of these accesses is a write. Since these races lead to unintended non-deterministic executions of the program, it is important to detect the races for effective debugging.

The previous tool [14] for detecting races in concurrent interrupt handlers converts each handler into a corresponding thread to use existing techniques that detect races in multi-threaded program. Such the techniques can be classified into static and dynamic techniques. Static analysis [9,12,13] consider the entire program and warn about potential races in all possible execution orders. However, these techniques tend to make conservative assumptions that lead to a large number of false positives. Therefore, the previous approach reports too many false positives without any false negative, because it uses a static technique for detecting races. On the other hand, dynamic techniques [1,6,16] report still less false positives than static techniques, but their coverage is limited to the paths and thread interleaving explored at runtime. In practice, the coverage of dynamic techniques can be increased by running more tests. In addition, dynamic data races detectors are severely limited by their runtime overhead.

There are two different methods for dynamic race detection in multi-threaded programs: post-mortem and on-the-fly methods. A post-mortem technique records events that occur during a program execution, and then analyzes or

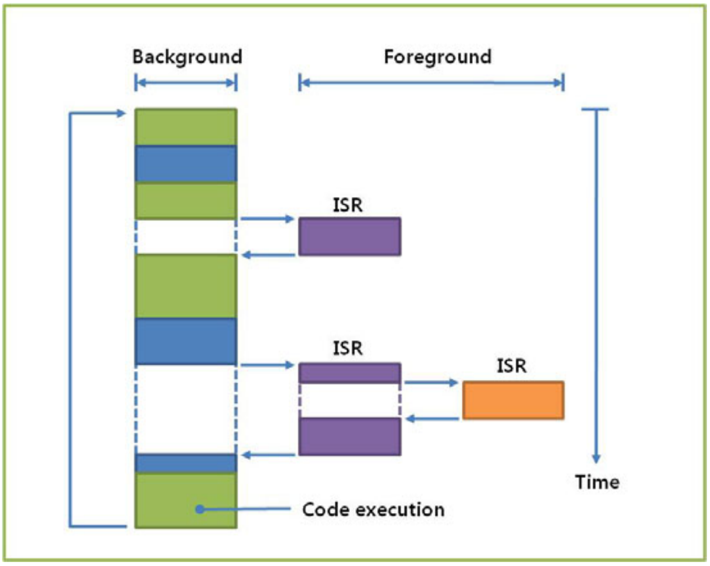


Fig. 1. A Dynamic Structure of Embedded Software

replays them after the program execution. An on-the-fly technique records and analyzes information during a program execution. This technique is based on *lockset* [4,6,15] or *happens-before algorithm* [2,4,11]. A lockset algorithm checks if two threads accessing a shared memory location hold a common lock. This algorithm is practically efficient, while it reports many false positives. A happens-before algorithm is based on Lamport's happens-before relation [8]. This algorithm may report still fewer false positives, while it incurs huge overhead in performance. Therefore, recent race detectors tend to combine happens-before techniques with lockset based ones to obtain the advantages of both algorithms.

3 A Program Conversion Tool

This paper presents a novel tool that converts an original program with concurrent interrupt handlers into the corresponding set of semantically equivalent POSIX thread [3] considering real-time scheduling policies and interrupt priorities of processor. Therefore, we detect the races in the converted programs using a dynamic detection tool for multi-treaded programs to reduce false positives. The dynamic tool is Helgrind+ that is an extension of Helgrind which combines the happens-before algorithm and the Lockset algorithm. Helgrind is a Valgrind tool [17] for detecting races in C, C++ and Fortran programs that use the POSIX threads. It uses an Eraser algorithm [15] which is improved based on the happens-before algorithm of VisualThreads [5] in order to reduce false positives.

Figure 2 shows the design of our tool which consists of three steps: a source scanner, a foreground conversion, and a background conversion. The original source code is scanned by source scanner module. The foreground one converts every interrupt handler into a POSIX thread using two modules: an interrupt handler exploration, and function pointer conversion. The interrupt handler exploration module explores the interrupt handlers using the *interrupt vector table*

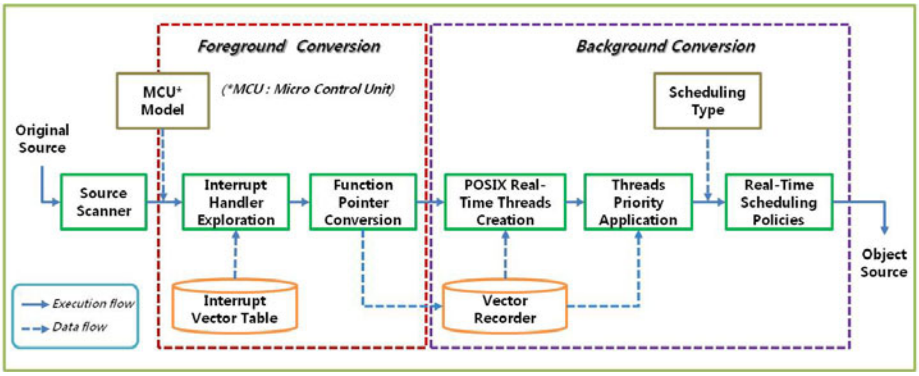
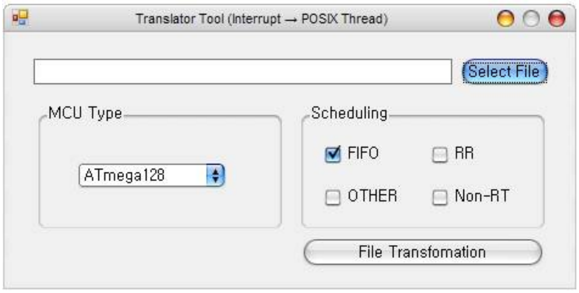

Foreground Conversion
Background Conversion
MCU* Model
(*MCU : Micro Control Unit)
Scheduling Type
Original Source
Source Scanner
Interrupt Handler Exploration
Function Pointer Conversion
POSIX Real-Time Threads Creation
Threads Priority Application
Real-Time Scheduling Policies
Object Source
Interrupt Vector Table
Vector Recorder
Execution flow
Data flow

Fig. 2. Design of Program Conversion Tool

that defines interrupt priorities of processor and function names used at conversion time. The function pointer conversion converts the explored interrupt handlers into the corresponding function pointers, and records the results into the interrupt handler table. These results are used as the parameters to call `pthread_create()` which creates POSIX threads for concurrent interrupt handlers. To handle interrupts that occur asynchronously, the each handler function is converted to be included into an infinite loop.

The background conversion consists of three modules to apply priorities and real-time scheduling: POSIX thread creation, thread priority application, and real-time scheduling policies. Interrupts are activated and deactivated using `sei()` and `cil()` functions, respectively, in the processor developed by ATmel which provides the WinAVR compiler. Therefore, the POSIX thread creation module converts every code which calls `sei()` function into one `pthread_create()` calling. We use only two of four parameters needed in the function: the thread identifier and the function that will automatically run at the end of the thread creation process using the information stored in the interrupt handler table. Also, the module deletes or annotates every code that calls the `cil()` function, because the function is of no use in POSIX threads. To execute the converted program under the same condition with that of the original source code, we apply the thread priorities and real-time scheduling in the last two steps of conversion. The thread priority application module changes attributes of threads using the *interrupt handler table.* The real-time scheduling module sets its policies. We use the PHREAD_EXPLICIT_SCHED option provided in POSIX real-time threads to change the policy. A list of optional policies is as follows.

(1) The `SCHED_FIFO` option allows a thread to run until another thread becomes ready with a higher priority, or until it blocks voluntarily. When a thread with this option becomes ready, it begins executing immediately unless a thread with equal or higher priority is already executing.
(2) The `SCHED_RR` option is much the same as SCHED_FIFO policy except that the running thread will be preempted so that the ready thread can be executed, if a thread is ready with SCHED_RR policy executes for more than a fixed period of the time slice interval without blocking, and another



MCU Type: ATmega128
Scheduling: [✓ FIFO] RR OTHER Non-RT

Fig. 3. User Interface of Program Conversion Tool

thread with SCHED_RR or SCHED_FIFO policy and the same priority. When threads with SCHED_FIFO or SCHED_RR policy wait on a condition variable or wait to lock a mutex, they will be awakened in priority order.

(3) The `SCHED_OTHER` option may be an alias for SCHED_FIFO, or it may be SCHED_RR, or it may be something entirely different. The real problem with this ambiguity is not that we do not know what SCHED_OTHER does, but that we have no way of knowing what scheduling parameters it might require.

4 Experimentation

Figure 3 shows the user interface of our tool that has been implemented in the C# language under the Windows operating system. Using the interface, the user selects the program to be debugged, the processor type, and the real-time scheduling policy to convert the program into a POSIX thread-based program.

To evaluate the accuracy of our tool, we use two programs with concurrent interrupt handlers for UAV flight control. These programs are parts of a UAV ground control station. The first program is the Knob Assembly Control (KAC) that is a firmware embedded into the micro controller unit of Knob assembly to execute autopilot commands by communicating with a real-time computer of the ground control station. KAC controls the altitude, the speed, the roll/heading and the azimuth angles of an aircraft. The part shown in dotted line in Figure 4 shows a Knob assembly. If the KAC gets commands to control aircraft by real-time computer, it outputs those values have been input by user using the Knob dials to the dot matrixes and the real-time computer. A micro controller unit communicates with a dot matrix using the Inter Integrated Circuit (I^2C) or the Two-wire Serial Interface (TWI), and communicates with the real-time computer using RS-232C. Commands or some feedback on the commands are transmitted

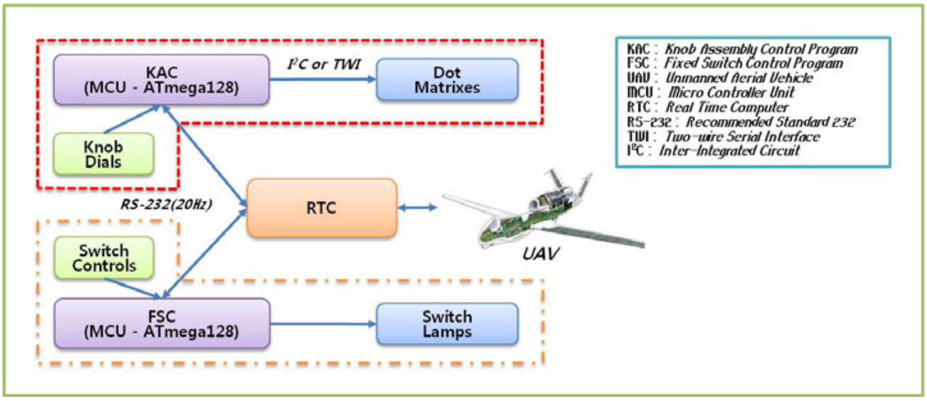

KAC (MCU - ATmega128)
I²C or TWI
Dot Matrixes
Knob Dials
RS-232(20Hz)
RTC
UAV
Switch Controls
FSC (MCU - ATmega128)
Switch Lamps
KAC : Knob Assembly Control Program
FSC : Fixed Switch Control Program
UAV : Unmanned Aerial Vehicle
MCU : Micro Controller Unit
RTC : Real Time Computer
RS-232 : Recommended Standard 232
TWI : Two-wire Serial Interface
I²C : Inter-Integrated Circuit

Fig. 4. Structure of KAC/FSC Program

on a cycle, 20Hz, by a timer interrupt and a serial communication interrupt. The second program is the Fixed Switch Control (FSC) that is also a firmware embedded in the micro controller unit of the Fixed Switch to control a parafoil or an engine for collecting an aircraft without runaway by communicating with the real-time computer. The part shown in dashed or dotted-line in Figure 4 represents the Fixed Switch. This program passes commands to the real-time computer and receives feedback from the real-time computer on a cycle. We assume that these programs are executed according to the scenarios denoted in Figure 5.

We translated these programs into thread-based programs using our tool implemented in C# language. We detect races in those programs using Helgrind+. We have performed experiments on Intel Pentium 4 with 2GB of RAM under Fedora 12 operating system. To empirically compare the accuracy, we installed Helgrind+ for dynamic race detection and Locksmith [13,14] for static race detection. We used the MSM-short (Memory State Machine for short-running application) option [1] provided in Helgrind+. This option is suitable for unit testing of program development or debugging of program with short execution time. We experimented for the race detection five times with each scheduling policy, because the result of race detection with Helgrind+ is affected by the real-time scheduling policy.

Figure 6 shows the result of race detection with Helgrind+ and Locksmith. The third and the fourth columns of Figure 6 show the number of races detected by Locksmith and Helgrind+, respectively. The fourth column is divided into four subcolumns according to the real-time scheduling policies: FIFO, RR, OTHER, and non-real time. In the KAC program, Locksmith detected one race toward each of ten shared variables. By source code analysis, we found that

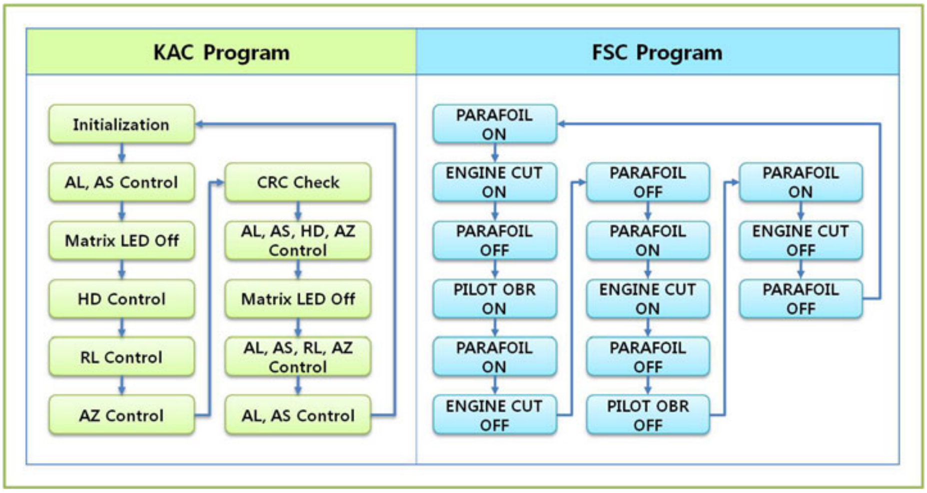


Fig. 5. Execution Scenario for Experimentation

Program	Result of Detection	Static Tool (Locksmith)	Dynamic Tool (Helgrind+)			
			FIFO	RR	OTHER	Non-RealTime
KAC	True Positives	8	3	1	3	6
	false positives	2	0	0	0	0
	false negatives	0	5	7	5	2
FSC	True Positives	5	2	2	3	3
	false positives	3	0	0	0	0
	false negatives	0	3	3	2	2

Fig. 6. Result of Data Race Detection

races detected toward two shared variables of them are false positives, because the source code involved in the races are executed once in a program execution to allocate memory before activating interrupt handlers. On the other hand, Helgrind+ detects one race toward each one of three, one, and three shared variables with FIFO, RR and OTHER, real-time scheduling, respectively. Also, the tool detects one race toward each one of six shared variables without any scheduling policy. Helgrind+ does not report a false positive. Thus, the results are different according to scheduling policies. The reason is that partial order executions of program depend on scheduling policies. In the FSC program, the two tools have also produced similar results.

5 Conclusion

This paper presents a program conversion tool that converts the concurrent interrupt handlers into semantically equivalent POSIX threads considering real-time scheduling policies and interrupt priorities of processor, and then detects the races in the converted programs using a dynamic detection tool, called Helgrind+, developed for multi-threaded programs.

By using two flight control programs of unmanned aerial vehicle, we were able to identify the existence of races in the embedded software with concurrent interrupt handlers. And the results of experiment show that the previous tool reports two and three false positives without any false negative in each software, while our tool does not report any false positive. However, our tool reports still more false negatives than the previous tool. There still remains more work which include additional effort to effectively improve the portability without among various hardware.

References

1. Jannesari, A., Bao, K., Pankratius, V., Tichy, W.F.: Helgrind+: An efficient dynamic race detector. In: Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–13. IEEE Computer Society Press, Washington, DC, USA (2009)
2. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A theory of data race detection. In: Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging, PADTAD 2006, pp. 69–78. ACM, New York (2006)
3. Butenhof, D.R.: Programming with posix threads. Addison-Wesley Professional (1997)
4. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. In: Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, PADD 1991, pp. 85–96. ACM, New York (1991)
5. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded java program test generation. In: Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande, JGI 2001. ACM, New York (2001)
6. Jannesari, A., Tichy, W.F.: On-the-fly race detection in multi-threaded programs. In: Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD, pp. 6:1–6:10. ACM, New York (2008)
7. Labrosse, J.J.: Microc/os-ii the real-time kernel, 2nd edn., pp. 32–66. CMP Books (2002)
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of ACM, 558–565 (1978)
9. Le, W., Yang, J., Soffa, M.L., Whitehouse, K.: Lazy preemption to enable path-based analysis of interrupt-driven code. In: Proceeding of the 2nd Workshop on Software Engineering for Sensor Network Applications, SESENA 2011, pp. 43–48. ACM, New York (2011)
10. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1, 74–88 (1992)
11. Park, S.H., Park, M.Y., Jun, Y.K.: A Comparison of Scalable Labeling Schemes for Detecting Races in OpenMP Programs. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 68–80. Springer, Heidelberg (2001)
12. Pessanha, V., Dias, R.J., Lourenço, J.A.M., Farchi, E., Sousa, D.: Practical verification of high-level dataraces in transactional memory programs. In: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2011, pp. 26–34. ACM, New York (2011)
13. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. SIGPLAN Not. 41, 320–331 (2006)
14. Regehr, J., Cooprider, N.: Interrupt verification via thread verification. Electron. Notes Theor. Comput. Sci. 174, 139–150 (2007)
15. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15, 391–411 (1997)
16. Tahara, T., Gondow, K., Ohsuga, S.: Dracula: Detector of data races in signals handlers. In: Asia-Pacific Software Engineering Conference, pp. 17–24 (2008)
17. Valgrind-project: Helgrind: a data-race detector (2007)