Bachelor Thesis

Robin Willenbrock

Static Detection of Data Races in Interrupt-Driven Software Using Reduced Inter-Procedural Control Flow Graphs

June 22, 2024

supervised by:
Ulrike Engeln

Hamburg University of Technology (TUHH)
Technische Universität Hamburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Contents

List of Figures

1 Introduction

2 Background

2.1 Interrupt-Driven Systems

An interrupt-driven system is an architectur where the flow of the execution is changed by unpredictable events in the system also known as interrupts. Interrupts can be caused by hardware devices, software conditions or external signals forcing the proscessor to suspend the current task to execute an interrupt handler or interrupt service routine(ISR). Interrupt-driven systems are used in real- time operating systems(RTOS), embedded systems and generally in systems where timely responses are nessesary [1].

The management of the interrupts to keep the fast responsivness of the system is the challenging part of an interrupt-driven system. Interrupts occur unpredictable so you have to consider every execution flow. To ensure the execution of critical interrupts, interrupts are often prioritized, so higher priority events can interrupt lower ones and get handled immediatly. When handling an interrupt the current state of the processos is saved and the context is switched to the ISR [1].

There are different types of interrupts based on the source they come from. There are Hardware interrupts, software interrupts and external interrupts. Each interrupt has different mechnisms and priorities, which influence the behavior of the system to the interrupt [4].

The unpredictivness and asynchronous nature of the interrupts bring a lot of challenges in designing and implementing an interrupt-driven system. One of the biggest challenges is the correct handling of high-priority interrupts without delaying them substantually. Which needs an sophisticated scheduling and prioritization mechanism. The execution of main programm and ISR needs to be handled properly to ensure data intergrity. Furthermore, handling context switches, preserving system state and avoiding deadlocks complitcate the development of an interrupt-driven system [5].

2.2 Reduced Inter-Procedural Control Flow Graphs (RIPCFG)

A Control Flow Graph (CFG) is a representation of all paths that might be traversed through a program during its execution. In the context of inter-procedural analysis, an Inter-Procedural Control Flow Graph (ICFG) extends this concept by incorporating the control flow between different procedures or functions in a program. Reduced Inter-Procedural Control Flow Graphs (RIPCFG) are optimized versions of ICFGs, designed to simplify the analysis while preserving essential information [2].

RIPCFGs utilize various techniques such as node merging, edge contraction, and the elimination of non-essential nodes to reduce the complexity of the graph without losing critical control flow information. These reduction techniques make it feasible to analyze large and complex software systems, which would be computationally prohibitive with full ICFGs. By retaining enough detail, RIPCFGs facilitate accurate static analysis, including data flow analysis, control flow analysis, and the detection of potential data races and other concurrency issues. The reduced size and complexity result in more efficient analysis algorithms, enabling faster detection and resolution of issues in the software [1].

2.2.1 Techniques for Constructing RIPCFGs

Several techniques are employed to construct RIPCFGs effectively. Node merging involves combining nodes that represent similar or redundant control flow paths, reducing the overall number of nodes in the graph. Edge contraction simplifies the graph by collapsing edges that do not significantly affect the control flow, thereby minimizing the number of connections between nodes. The elimination of non-essential nodes focuses on removing nodes that do not contribute to the primary control flow, such as nodes representing trivial or inline functions. These techniques collectively enhance the scalability and efficiency of static analysis, making it practical to analyze complex software systems [6].

2.2.2 Applications of RIPCFGs

RIPCFGs are widely used in various static analysis applications, including data flow analysis, control flow analysis, and the detection of concurrency issues such as data races and deadlocks. By providing a simplified yet accurate representation of the program's control flow, RIPCFGs enable efficient analysis of large codebases, facilitating the identification and resolution of potential issues early in the development process. Additionally, RIPCFGs are instrumental in optimizing compilers, where they assist in optimizing code by identifying redundant or inefficient paths and enabling more effective code generation strategies [6].

2.3 Data Races

Data races occur when two or more threads access shared data concurrently, and at least one of the accesses is a write. This condition can lead to unpredictable and erroneous behavior, making the detection and resolution of data races a critical aspect of concurrent programming [3].

In systems where multiple threads execute without proper synchronization, data races arise from conflicting operations on shared data. The outcome of a program with data races is non-deterministic, as the order of execution and interleaving of operations can vary, leading to inconsistent results and hard-to-reproduce bugs.

2.3.1 Types of Data Races

There are primarily two types of data races. Write-write races occur when multiple threads write to the same variable simultaneously, potentially causing data corruption as the final value of the variable depends on the order of writes, which is non-deterministic. Read-write races occur when one thread reads a variable while another thread writes to it concurrently, leading to the possibility of reading stale or inconsistent data, which can cause erroneous program behavior [3].

2.3.2 Detection Techniques

Various static and dynamic analysis techniques are used to detect data races. Static analysis involves examining the code without executing it. Tools such as RacerX [2] analyze the program's source code to detect race conditions and deadlocks by simulating different execution paths. Static analysis can identify potential data races by examining the control flow and data dependencies in the code, enabling early detection and resolution of issues.

Dynamic analysis, on the other hand, monitors the program during execution to identify potential race conditions. FastTrack [3] is a dynamic analysis tool that efficiently detects data races by maintaining a happens-before relationship among the threads' operations and checking for violations. Dynamic analysis provides a runtime perspective, capturing actual execution scenarios and detecting data races that may not be evident through static analysis alone.

Hybrid techniques combine static and dynamic analysis to leverage the strengths of both approaches. Static analysis can identify potential race conditions, which can then be confirmed or refuted by dynamic analysis during actual program execution. This approach allows for comprehensive detection and resolution of data races, ensuring both theoretical and practical coverage of potential issues.

2.3.3 Implications of Data Races

The presence of data races in a program can lead to several critical issues. The outcome of a program with data races is non-deterministic, making it difficult to reproduce and debug errors. Concurrent writes to shared data without proper synchronization can lead to data corruption, resulting in incorrect program behavior and potentially causing system crashes. Furthermore, data races can be exploited by malicious actors to create security vulnerabilities. Inconsistent data states can be manipulated to bypass security checks or corrupt sensitive data, posing significant risks to system integrity and security [7].

2.3.4 Strategies for Preventing Data Races

Preventing data races requires careful design and implementation of concurrent programs. One effective strategy is to use proper synchronization mechanisms, such as mutexes, semaphores, and condition variables, to control access to shared data. These mechanisms ensure that only one thread can access the shared data at a time, preventing conflicting operations. Avoiding shared mutable state is another effective strategy, where threads operate on local copies of data instead of shared data, reducing the potential for conflicts. Designing thread-safe data structures and algorithms that inherently manage concurrent access also helps prevent data races, ensuring reliable and predictable program behavior [8].

2.4 Static Detection of Data Races in Interrupt-Driven Systems

In interrupt-driven systems, the asynchronous nature of interrupts and the concurrent execution of ISRs and the main program introduce significant challenges in ensuring data consistency and detecting data races. Static analysis techniques, particularly those using RIPCFGs, offer a promising approach to identifying potential data races without the need for exhaustive testing or runtime monitoring [1].

The approach involves constructing RIPCFGs for the program, including both the main code and ISRs, capturing the control flow and potential interactions between them. By analyzing the RIPCFGs, paths where shared data is accessed concurrently without proper synchronization can be identified, indicating potential data races. Integrating the static analysis tool with the development workflow enables continuous detection and resolution of data races during the software development lifecycle, improving the reliability and correctness of interrupt-driven systems [1].

2.4.1 Methodology for Static Detection

The methodology for static detection of data races in interrupt-driven systems involves several key steps. First, the RIPCFGs are constructed for the entire program, including the main code and ISRs. This involves analyzing the control flow and identifying points where the main program and ISRs interact with shared data. Next, the RIPCFGs are analyzed to identify potential race conditions, focusing on paths where concurrent access to shared data may occur without proper synchronization. Finally, the analysis results are integrated with the development workflow, allowing developers to address identified race conditions early in the development process. This methodology ensures comprehensive coverage of potential race conditions and facilitates timely resolution of issues [1].

3 Implementation

3,5 Wochen

4 Evaluation

1,5 Woche

5 Conclusion

Indroduction+Conclusion und Allgemeine Überarbeitung 0,5 Woche 1 Wochen Korrekturlesen und Einarbeitung =9 Wochen bei Vollarbeitszeit an BE

Bibliography

Lightweight Data Race Detection for Production by Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, Benjamin P. Wood

A Deployable Sampling Strategy for Data Race Detection by Yan Cai1, Jian Zhang, Lingwei Cao, and Jian Liu

Bibliography

[1] Wang, Y., Gao, F., Wang, L., Yu, T., Zhao, J., & Li, X. (2020). Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software. IEEE Transactions on Software Engineering.

[2] Engler, D., & Ashcraft, K. (2003). RacerX: Effective, Static Detection of Race Conditions and Deadlocks. ACM SIGOPS Operating Systems Review.

[3] Flanagan, C., & Freund, S. N. (2009). FastTrack: Efficient and Precise Dynamic Race Detection. ACM SIGPLAN Notices.

[4] Burns, A., & Wellings, A. (2009). Real-Time Systems and Programming Languages. Addison-Wesley.

[5] Labrosse, J. J. (2002). MicroC/OS-II: The Real-Time Kernel. CMP Books.

[6] Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.

[7] Adve, S. V., & Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial. IEEE Computer.

[8] Herlihy, M., & Shavit, N. (2008). The Art of Multiprocessor Programming. Morgan Kaufmann.