

Mälardalen University Licentiate Thesis  
No. 24

# Deterministic Replay Debugging of Embedded Real-Time Systems using Standard Components

Daniel Sundmark

March 2004



**MÄLARDALEN UNIVERSITY**

Department of Computer Science and Engineering  
Mälardalen University  
Västerås, Sweden

Copyright © Daniel Sundmark, 2004  
ISBN 91-88834-35-2  
Printed by Arkitektkopia, Västerås, Sweden  
Distribution: Mälardalen University Press

## Abstract

Men and women make mistakes. They always have and they always will. Naturally, software engineers are no exception to this rule. When software engineers make their mistakes, these manifest in the form of buggy software. Luckily, men and women often strive to correct the mistakes they make. In software engineering, this process is called debugging.

In simple sequential software, debugging is fairly easy. However, in the realm of embedded real-time software, debugging is made significantly harder by factors such as dependency of an external context, pseudoparallelism or true parallelism, and other real-time properties. These factors lead to problems with execution behavior reproducibility. When a failure is discovered, we need to be able to reproduce this failure in order to examine what went wrong. If the erroneous behavior cannot be reproduced, we will not be able to examine the process leading to the failure.

Previous work has proposed the use of execution replay debugging in order to solve this problem. Execution replay is a general term for a set of methods to record system behavior during execution and to use these recordings in order to reproduce this behavior during debugging sessions. This way, we may achieve a reproducible execution behavior for non-deterministic systems. Historically, many replay methods have been highly platform-dependent, craving specialized hardware, operating system or compilers.

In this thesis, we describe a replay method, called Deterministic Replay, able to run on top of standard components. We also describe the Time Machine, which is the implementation of the Deterministic Replay method. Further, we give an in-depth description of the method for pinpointing interrupts used by the Time Machine. In addition, we present results from two case studies where the Deterministic Replay method was incorporated into two full-scale industrial real-time systems. These results show that our method of debugging multi-tasking real-time systems not only is applicable in industrial applications, but also that it can be introduced with little effort and small costs regarding application performance.

**Keywords:** debugging, replay, real-time systems, embedded systems



*To Kristina*



## Acknowledgements

This work has been supported by the KK Foundation (KKS) and Mälardalen University.

I would like to thank everybody at the department for creating a highly enjoyable working site. Especially, I would like to thank my fellow PhD students Anders Pettersson and Joel Huselius for taking the time to answer my questions and to my supervisors Henrik Thane and Hans Hansson for helping me row this boat ashore. Furthermore, I would like to thank Thomas Nolte for his great support, Jonas Neander for keeping the positive spirit up and Jocke Fröberg for many interesting discussions. Many thanks to my fellow lecturers Dag and Jukka. We've had many laughs and I think we give a great course. Thanks also to Mic and professors Mats, Ivica, Christer and Lars for valuable input and support. In addition, I would like to thank Aje, Markus, Micke, Ralf, Leif, Manne, Johan and everybody in the floorball crew, Roger and the other discgolfers, and Radu, Lennvall, Damir and the soccer guys. You make the hard days seem shorter and the good days seem longer. Thanks also to Möller and Andreas and to Katrin for keeping our house clean. Not least, I would like to thank Harriet, Monica and Malin. We would be lost in space without you.

Naturally, I owe a lot to my family and friends. Thank you Mom and Dad for your love and support. Thanks to Anneli, Niclas, Gösta, Margot, Janne, Git-tan, Micke, Sara, Barbro, Sigurd, Helene, Calle, Victoria and Lovisa. Special thanks to Grabbarna Grus: Boberg, Arnholm, Loffe, David, Wier and Gurra. If anyone has enriched my working hours, it's you. Compadres de luxe! Thanks also to stugängänget and to Mattias Sjögren for all valuable help and companionship during my undergraduate studies. In addition, I would like to thank sensei Enrico and everybody at the dojo.

And, of course, Kicki, you have been there for me through thick and thin. I love you.

Muchas Gracias!

Västerås, February 2004

Karl Daniel Sundmark





# Contents

<b>Contents</b>	<b>vii</b>
<b>List of Publications</b>	<b>xi</b>
<b>I Thesis</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Basic Concepts . . . . .	4
1.1.1 Bugs . . . . .	4
1.1.2 Real-Time Systems . . . . .	4
1.1.3 Testing and Debugging . . . . .	4
1.2 Problem Statement . . . . .	5
1.3 Outline . . . . .	6
<b>2 Contributions</b>	<b>7</b>
2.1 Paper A (Chapter 4) . . . . .	7
2.2 Paper B (Chapter 5) . . . . .	8
2.3 Paper C (Chapter 6) . . . . .	8
2.4 Paper D (Chapter 7) . . . . .	9
<b>3 Conclusions</b>	<b>11</b>
3.1 Summary . . . . .	11
3.2 Future Work . . . . .	11

<b>II</b>	<b>Included Papers</b>	<b>13</b>
<b>4</b>	<b>Paper A: Replay Debugging of Embedded Real-Time Systems: A State of the Art Report</b>	<b>15</b>
4.1	Introduction . . . . .	17
4.1.1	Code Inspection . . . . .	18
4.1.2	Diagnostic Output . . . . .	18
4.1.3	Cyclic Debugging . . . . .	19
4.1.4	Static Analysis . . . . .	20
4.2	Complex System Debugging Issues . . . . .	20
4.2.1	Determinism and Reproducibility . . . . .	20
4.2.2	Context Issues . . . . .	22
4.2.3	Ordering Issues and Concurrency . . . . .	23
4.2.4	Timing Issues . . . . .	27
4.2.5	Embedded Systems . . . . .	29
4.2.6	Complex Debugging Summary and Problem Statement	30
4.3	Debugging by Execution Replay . . . . .	33
4.3.1	Replay Debugging of Concurrent Systems . . . . .	33
4.3.2	Real-Time Systems Replay Debugging . . . . .	35
4.3.3	Asynchronous Events Reproduction . . . . .	35
4.3.4	On-The-Fly Race Detection . . . . .	40
4.4	Instrumentation for Replay . . . . .	41
4.4.1	The Probe Effect . . . . .	42
4.4.2	Instrumentation Jitter . . . . .	43
4.5	Deterministic Replay . . . . .	44
4.5.1	Reproducing System-Level Control Flow . . . . .	45
4.5.2	Reproducing External- and Internal Data Flow . . . . .	45
4.6	Replaying Long-Running Applications . . . . .	45
4.6.1	Starting a replay execution . . . . .	46
4.6.2	Checkpointing . . . . .	46
4.7	Related Research Projects . . . . .	47
4.8	Conclusions . . . . .	48
<b>5</b>	<b>Paper B: Replay Debugging of Real-Time Systems Using Time Machines</b>	<b>53</b>
5.1	Introduction . . . . .	55
5.1.1	Debugging Sequential Real-Time Programs . . . . .	55
5.1.2	Debugging Multi-Tasking Real-Time Programs . . . . .	56
5.1.3	Debugging by the Use of Time Machines . . . . .	56

---

5.1.4	Contribution . . . . .	57
5.2	Related Work . . . . .	58
5.3	The System Model . . . . .	58
5.4	The Mechanisms of the Time Machine . . . . .	59
5.4.1	What to Record . . . . .	60
5.4.2	How to Record . . . . .	62
5.4.3	The Historian . . . . .	63
5.4.4	Requirements on a Starting Point for the Replay Execution . . . . .	64
5.4.5	The Time Traveller . . . . .	66
5.5	Industrial Case Study . . . . .	68
5.6	Future Work . . . . .	68
5.7	Conclusions . . . . .	69
<b>6</b>	<b>Paper C: Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies</b> . . . . .	<b>73</b>
6.1	Introduction . . . . .	75
6.1.1	Contribution . . . . .	75
6.1.2	Paper Outline . . . . .	75
6.2	Background and Motivation . . . . .	75
6.2.1	Replay Debugging . . . . .	76
6.2.2	Real-Time System Debugging using Time Machines and Deterministic Replay . . . . .	76
6.2.3	ABB Robotics System Model . . . . .	77
6.2.4	SAAB Avionics System Model . . . . .	79
6.3	Technique Implementations . . . . .	79
6.3.1	VxWorks Instrumentation . . . . .	79
6.3.2	ABB Robotics Instrumentation . . . . .	81
6.3.3	SAAB Avionics Instrumentation . . . . .	82
6.3.4	Time Machine . . . . .	82
6.3.5	IDE and Target System Integration . . . . .	84
6.4	Benchmark . . . . .	86
6.4.1	ABB Robotics . . . . .	86
6.4.2	SAAB Avionics . . . . .	87
6.5	Conclusions . . . . .	87
6.6	Future Work . . . . .	88

---

<b>7</b>	<b>Paper D: Pinpointing Interrupts in Embedded Real-Time Systems using Context Checksums</b>	<b>93</b>
7.1	Introduction . . . . .	95
7.1.1	Background . . . . .	95
7.1.2	Related Work . . . . .	96
7.1.3	Problem Formulation . . . . .	98
7.1.4	Contribution . . . . .	98
7.1.5	Paper Outline . . . . .	98
7.2	Context Checksums . . . . .	98
7.2.1	Execution Context . . . . .	99
7.2.2	Register Checksum . . . . .	99
7.2.3	Stack Checksum . . . . .	100
7.2.4	Partial Stack Checksum . . . . .	101
7.3	Approximation Accuracy . . . . .	102
7.4	Simulation . . . . .	105
7.4.1	Approximation Accuracy . . . . .	106
7.4.2	Perturbation . . . . .	109
7.5	Conclusions . . . . .	110
7.6	Future Work . . . . .	111

# List of Publications

The following articles are included in this licentiate<sup>1</sup> thesis:

- A. *Replay Debugging of Embedded Real-Time Systems: A State of the Art Report*, Daniel Sundmark, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-156/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February 2004.
- B. *Replay Debugging of Real-Time Systems using Time Machines*, Henrik Thane, Daniel Sundmark, Joel Huselius and Anders Pettersson, In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD), pages 288 – 295, Nice, France, April 2003.
- C. *Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies*, Daniel Sundmark, Henrik Thane, Joel Huselius, Anders Pettersson, Roger Mellander, Ingemar Reiyer and Mattias Kallvi, In M. Ronsse, K. De Bosschere (eds), proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging (AADEBUG), pages 211 – 222, CComputer Research Repository<sup>2</sup>, Gent, Belgium, September 2003.
- D. *Pinpointing Interrupts in Embedded Real-Time Systems using Context Checksums*, Daniel Sundmark and Henrik Thane, A version of this paper has been submitted for publication.

---

<sup>1</sup>A licentiate degree is a Swedish graduate degree halfway between M.Sc. and Ph.D.

<sup>2</sup><http://www.acm.org/corr/>

Besides the above articles, I have (co-)authored the following scientific papers:

- I. *The Asterix Real-Time Kernel* Henrik Thane, Anders Pettersson and Daniel Sundmark, In Proceedings of the Thirteenth EUROMICRO International Conference on Real-Time Systems, Industrial Session, Technical University of Delft, Delft, The Netherlands, June 2001.
- II. *Debugging using Time Machines: replay your embedded system's history*, Henrik Thane and Daniel Sundmark, In Proceedings of Real-Time & Embedded Computing Conference, Chapter 22, Milan, Italy, November 2001.
- III. *Starting Conditions for Post-Mortem Debugging using Deterministic Replay of Real-Time Systems*, Joel Huselius, Daniel Sundmark and Henrik Thane, In Proceedings of the Fifteenth EUROMICRO Conference on Real-Time Systems (ECRTS03), pages 177 – 184, Porto, Portugal, July 2003.
- IV. *Availability Guarantee for Deterministic Replay Starting Points in Real-Time Systems*, Joel Huselius, Henrik Thane and Daniel Sundmark, In M. Ronsse, K. De Bosschere (eds), proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging (AADE-BUG), pages 261 – 264, Computer Research Repository<sup>3</sup>, Ghent, Belgium, September 2003.

---

<sup>3</sup><http://www.acm.org/corr/>

**I**  
**Thesis**





# Chapter 1

## Introduction

Software engineering is hard. In fact, it is so hard that the annual cost of software errors is estimated to range from \$22.2 to \$59.5 billion per year in the US alone [1]. Few other engineering disciplines display such bleak figures. Why is this? Are software engineers simply more incompetent than other engineers or is there another explanation behind the multiplicity of undeliverable software produced by these individuals. One alternative explanation would be that software is discrete in its nature, making it impossible to interpolate between test results. A bridge that withstands a load of 20 tons could easily be assumed to withstand a load of 14, 15 or 17 tons. However, if you bear with us and assume a bridge built in software, a test showing that it withstands 20 tons would not guarantee that it could handle 14, 15 or 17 tons (or even 20 pounds for that matter). In addition, if the software bridge would fail to withstand a load, it is very hard to foresee in what fashion it would fail. Sure, it could collapse, but it could also implode, move left or fall up in the sky. Software is pure abstraction and is not bound by the laws of physics.

Yet another explanation of the poor success ratio of software projects is the fact that software engineering is a fairly young engineering discipline. Methods and tools for aiding developers in their task are still in an early phase of their development. This fact makes software engineering more arbitrary than other engineering disciplines and the process of development is often performed at the engineer's own discretion. This is the situation we intend to improve by our contributions.

## 1.1 Basic Concepts

In order to further describe our results, we will need to establish a few basic concepts.

### 1.1.1 Bugs

“It is easy to talk about ‘bug-free’ software, but because it is nonexistent, we focus on what does exist – buggy software”, Telles and Hsieh [2].

A software bug is a defect in a program, that may (or may not) cause the program to fail, depending on whether the bug is allowed to infect the program execution or not. Bugs may be very simple (e.g. erroneous variable assignments) or rather complex (e.g. improperly synchronized shared-variable accesses). In this thesis, we address both simple and complex bugs, but since the latter are significantly harder to find and remove, this is where something really has to be done.

### 1.1.2 Real-Time Systems

Traditionally, a real-time system is a system whose correctness not only depends on its ability to produce correct results, but also on the ability of producing these results within a well-defined interval of time. Textbook examples of such systems are ABS braking systems in cars, factory automation systems or flight control systems.

### 1.1.3 Testing and Debugging

In software engineering, *testing* is the process of revealing the existence of bugs in a program, whereas *debugging* is the process of finding them and removing them. To aid the developers in the debugging process, debugger tools, in which an erroneous program can be thoroughly examined during run-time, are often used. However, in order for an erroneous execution of a program to be examined, the behavior of the program needs to be reproducible. For sequential software with no real-time requirements, execution reproducibility is trivial. If such a program is started a number of times with the same input, all executions will exhibit identical behavior and produce identical output. Unfortunately, very few systems developed in today’s industry conform to the strict requirements of non-real-time sequential software. As the behavior of real-time multi-tasking software is inherently harder to reproduce, regular debug-

ging methods cannot be used to track down bugs in these systems. Therefore, methods have been proposed for recording events and data during run-time of otherwise non-reproducible systems and using this information to force identical execution behavior upon subsequent executions, more suited for thorough investigation. These methods are called *replay*-techniques.

## 1.2 Problem Statement

As stated in the previous section, reproducing the execution behavior of complex multi-tasking real-time systems is a non-trivial problem. This fact, combined with a lack of proper general debugger tools for these systems has led to a situation where system-integration-level and post-deployment bugs are often very hard to track down and remove, making them very costly to handle [1].

Replay methods for recording and reproducing system behavior has been proposed, but drawbacks in the form of specialized hardware requirements, significant performance penalties and non-general solutions have resulted in a situation where all hitherto proposed methods have been considered academic artefacts with no real commercial significance to the industrial sector. The aim of our research project, and thus of this licentiate thesis, is to remedy this situation and to make complex system debugging by deterministic replay an applicable industrial, as well as an academically accepted debugging method. In short:

*We aim at achieving the same level of reproducibility in multi-tasking real-time systems as the one we have in non-real-time sequential software.*

In order not to propose yet another academic artefact, we seek to build our method upon standard components, such as state-of-the-practice commercial debuggers, compilers, development environments and real-time operating systems. In addition, the method we propose should be portable and feasible to use in already existing applications.

The results presented in this licentiate thesis proposal are achieved within the scope of the DEBUG project at Mälardalen University, and are based on the Deterministic Replay method first proposed by Thane and Hansson in 2000 [3].

### **1.3 Outline**

The remainder of this thesis will be organized as follows: Chapter 2 will shortly describe the technical contributions of this thesis. In Chapter 3, we will conclude the first part of the thesis. In the second part of the thesis, included papers constitute Chapters 4 through 7.

## Chapter 2

# Contributions

In this chapter, the main technical contributions of each included paper are presented.

### 2.1 Paper A (Chapter 4)

Daniel Sundmark, *Replay Debugging of Embedded Real-Time Systems: A State of The Art Report*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-156/-2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February 2004

**Summary** This state of the art report surveys the current state of embedded real-time system debugging. We start out by giving a brief history of software debugging. Then, we describe the problems of embedded real-time system debugging by making the transition from regular sequential software debugging through multi-tasking software debugging and to embedded real-time system debugging. Apart from describing related work in complex system debugging, this report also describes the basics of Deterministic Replay, the method upon which this thesis is based.

**My contribution** I am the sole author of this paper.

## 2.2 Paper B (Chapter 5)

Henrik Thane, Daniel Sundmark, Joel Huselius and Anders Pettersson, *Replay Debugging of Real-Time Systems Using Time Machines*, In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Workshop, Nice, France, April 2003. This paper has also been submitted as invited journal publication.

**Summary** In this paper we present a new approach to Deterministic Replay using standard components. Our method facilitates cyclic debugging of real-time systems with industry standard real-time operating systems using industry standard debuggers. The method is based on a number of new techniques: A new marker for differentiation between loop iterations (as well as subroutine calls) for deterministic reproduction of interrupts and task preemptions, an algorithm for finding well-defined starting points of replay sessions, as well as a technique for using conditional breakpoints in standard debuggers to replay the target system. We also propose and discuss different methods for deterministic monitoring, and provide benchmarking results from an industrial case study demonstrating the feasibility of our method.

**My contribution** This paper is a joint effort. I wrote Section 5.4, discussing the Time Machine.

## 2.3 Paper C (Chapter 6)

Daniel Sundmark, Henrik Thane, Joel Huselius, Anders Pettersson, Roger Mellander, Ingemar Reiyer and Mattias Kallvi, *Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies*, In Proceedings of Fifth International Workshop on Automated and Algorithmic Debugging (AADEBUG) Gent, Belgium, September 2003.

**Summary** This paper describes a major Deterministic Replay debugging case study performed on a full-scale industrial robot control system, as well as a minor replay instrumentation case study performed on a military aircraft radar system. In this article, we show that replay debugging is feasible for complex multi-million lines of code software running on top of off-the-shelf real-time operating systems. Furthermore, we discuss how replay debugging can be introduced in existing systems without unfeasible analysis efforts. In addition, we present benchmarking results from both studies, indicating that the instrumentation overhead is acceptable and affordable.

---

**My contribution** The case studies described in this paper were carried out by all authors. However, I wrote the paper under supervision of, and in discussion with, the other authors.

## 2.4 Paper D (Chapter 7)

Daniel Sundmark and Henrik Thane, *Pinpointing Interrupts in Embedded Real-Time Systems using Context Checksums*, A version of this paper has been submitted for publication.

**Summary** When trying to reproduce execution behavior of a system affected by occurrences of interrupts, it is imperative that we are able to deduce exactly where these interrupts occur. This paper proposes the use of a checksum method for pinpointing the location of occurrence of interrupts. The checksum serves as an approximation of a unique program location marker and is based on the contents of the register set or the stack of the program preempted by the interrupt. This method is novel in that it neither requires any specialized hardware nor incurs any significant overhead on the system.

**My contribution** This paper is written by me, under supervision of my supervisor. I have done most of the work concerning the simulations. The work is based on an idea by my supervisor.





## **Chapter 3**

# **Conclusions**

### **3.1 Summary**

In this thesis, we have extended the Deterministic Replay method for embedded real-time systems debugging. We have shown that Deterministic Replay can be implemented using state of the practice standard components and that it performs well in the context of full-scale industrial real-time systems. In addition, we have examined the levels of execution time perturbation introduced by the instrumentation probes necessary for performing a Deterministic Replay debugging.

### **3.2 Future Work**

In Paper 6 we present results from two industrial case studies. These studies have been very valuable when establishing the feasibility and improving the generality of our method. In order to further enhance the generality of the Deterministic Replay method, additional case studies should be carried out. In addition, to be able to deterministically replay executions heavily loaded with interrupts, we will need to further improve the levels of accuracy for the context checksum unique markers.

We will also look into the possibility of using the data flow analysis of abstract interpretation in order to automatically identify which parts of the program state that should be recorded.

# Bibliography

- [1] NIST Report. The economic impacts of inadequate infrastructure for software testing., May 2002.
- [2] M. Telles and Y. Hsieh. *The Science of Debugging*. The Corolis Group, 2001.
- [3] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.

## **II**

# **Included Papers**



## **Chapter 4**

# **Paper A: Replay Debugging of Embedded Real-Time Systems: A State of the Art Report**

Daniel Sundmark

MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-156/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February 2004

### **Abstract**

Testing and debugging are major parts of a software development project, counted in time and money, as well as importance. As it is not likely that programmers and designers will make a sudden turn to start producing totally error-free designs or implementations, the test- and debug strategy of a software project will have a large influence over the overall quality of the end product. Over the years, massive resources have been spent in order to improve the quality of software. Recent reports suggest that multi-billion dollar amounts are spent each year on software maintenance in the U.S. alone. This in a time when nearly all software, desktop or embedded, grows increasingly more complex. Unfortunately, we have come to a point where many of the available tools for testing and debugging are insufficient for today's full-scale commercial software. This state of the art report surveys the available commercial and academic tools and methods for complex real-time software debugging.

## 4.1 Introduction

Ever since computer legend Dr. Grace Murray Hopper found a dead moth in the inner workings of the Mark II while looking for the cause of recent errors, the term *debugging* has been used for denoting the process of revealing and removing the causes of errors in computers and software. A classic debugging process starts with the discovery of a failure. The observed behaviour of an investigated system does not comply with the system specifications, meaning that at some point in time, prior to the observed propagation of the error, a bug has been executed, infecting the state of the system. When this infection violates the specifications, the system failure is observed. The process of dynamically observing system behaviour in order to find system specification violations is denoted *testing*. In software (and hardware) engineering, testing and debugging are very tightly coupled, since traditional debugging is impossible without testing (you can not remove a bug if you do not know that it exists) and testing is pointless without debugging (if a failure is observed and its cause not removed, we have accomplished very little).

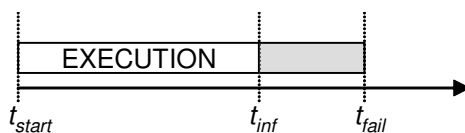


Figure 4.1: Infection, incubation time and fault propagation.

Now, we have come across a violation of the system specifications and are left with the actual debugging task of finding the cause of the failure and removing it. However, finding a bug and observing its propagation to output are two completely different things. Some software bugs, like trying to write without access to a protected memory area, will often propagate instantly. Other bugs, such as assigning an erroneous value to a variable, will infect the system, but might never cause the system to fail. For example, consider the timeline in Figure 4.1. The interval between the infection of the system  $t_{inf}$  (the execution of the bug) and the propagation of the bug to output  $t_{fail}$  is shaded in the figure. In an effort to follow the medical naming paradigm we adapted so far, we denote this interval *incubation time* [12]. In an ideal debugging process, we start out at the fault propagation and, moving backwards in time along the incubation period, we narrow down the functional and temporal domain within

which the infection of the system might have occurred. If we are successful, we will end up in  $t_{inf}$ , pointing at the bug.

Although today there exist several tools and methods for helping programmers and system developers in this process, testing and debugging consume a massive percentage of the time and money spent on software engineering projects. Debugging is highly manual detective labour and some malignant bugs might take months to track down, even by expert programmers using state-of-the-practice debugger tools. Other bugs are never found. There are fully operational safety-critical systems in use today with known, but never found, bugs. This situation is highly problematic, but, as in the old saying “Necessity is the mother of invention”, these difficulties have given rise to several efforts aimed at developing new and more efficient tools and methods for debugging.

But let us start from the beginning.

#### 4.1.1 Code Inspection

Intuitively, when something of your own design proves faulty, you try to investigate the cause of the failure. That day when the moth flew in through an open window at Harvard University and into the Mark II, short circuiting one of its relays, it caused the computer to produce erroneous outputs. The spontaneous reaction of Dr. Hopper was to open up the computer and check its internal electronic devices for “bugs”. The analogy in software development would be to inspect the source (or machine) code in a line by line fashion for implementation errors. For a long period of time, code inspection was state-of-the-practice debugging in software development. However, inspection of code is a strictly static debugging practice. If variables are used in the program, all possible values of these variables must be considered when analyzing the code. A more dynamic debugging technique where the actual program is executed might be able to tell us what variable values result in erroneous executions.

#### 4.1.2 Diagnostic Output

As computer programs became able to produce intermediate output, rather than being based on the old functional paradigm of taking an input, performing some calculation and finishing by producing an output, programmers soon realized that these intermediate outputs could be used for program diagnostics. Erroneous programs can be instrumented to produce output at strategic points in their execution, helping the programmer to track down bugs. In a less formal



language, this kind of debugging is often referred to as *printf-debugging* due to the massive use of the standard C library function `printf` to produce the diagnostic outputs.

### 4.1.3 Cyclic Debugging

However, the late moth of Mark II was found in the middle of the twentieth century and many things have happened since. Although diagnostic output and code inspection debugging strategies still are widely used (after all, each time you look at a piece of code, you evaluate its functional correctness), there are new methods as well. The most commonly used tool-based debugging strategy today is *cyclic debugging*, where erroneous programs are repeatedly re-executed and investigated in a specialized tool, enigmatically called the *debugger*.

In a sense, debugging using a debugger is not that different from regular code inspection. When using a debugger, we are still focusing on the examination of the source code. On the other hand, cyclic debugging is a dynamic method and using a debugger, we are not unaware of variable assignments and program branch selections. The role of the debugger tool is to visualize the behaviour of the program during its execution. When a program is started in a debugger, it is possible to follow its execution line by line in the source code. To make the connection between the running program and the source code of the program, compiler vendors provided the generation of a file, containing the symbol information gathered when parsing the code. This file could then be used to map the symbols to actual memory addresses in the program, which allowed users to follow the states of variables during the program execution. Debuggers providing this feature are called *symbolic debuggers* [28]. Practically all software debuggers used today are symbolic debuggers.

Another common denominator for all debuggers is the inclusion of two basic functions. These functions are *breakpointing* and *single-stepping*. Due to the fairly impossible task of keeping up with source code path inspection when debugging programs in real-time, it is essential that programmers are able to halt the execution of the program under investigation. Breakpointing enables the programmer to place markers (breakpoints) at arbitrary locations in the source code. As these breakpoints are encountered by the program, the debugger tool stops the execution, making a thorough state examination of the program possible. Once the program is in a halted state, the programmer is able to manually single-step through the execution. After each step, the execution is halted once again. Single stepping can be performed on an instruction-by-

instruction- or an function-by-function basis and often proves to be very useful when trying to investigate the progress of an execution and when trying to track down variable assignments.

Apart from the next section, the remainder of this report will discuss techniques based on cyclic debugging using debugger tools.

#### 4.1.4 Static Analysis

As we mentioned code inspection in an above section, we should also mention static analysis as a means of debugging programs. If testing is the process of revealing the presence of bugs in a program and debugging is the process of tracking down these bugs, static analysis can be said to be a hybrid testing- and debugging technique. Static analysis is basically a general term for different automated code inspection methods. These methods can be applied to programs with known bug presences, but also to programs without known bugs, just as an assurance of robustness or as a means of ensuring a certain quality of a program.

Static analysis tools can perform simple source code analysis, such as looking for uninitialized variables or dangling pointers, but also more complex analysis, such as searching for potentially malignant shared memory accesses in concurrent programs [27].

## 4.2 Complex System Debugging Issues

So, what is the problem? Apparently, several highly practical debugger methods and tools are available today for a reasonable price (in some cases even for free). What does today's state-of-the-practice and state-of-the-art debuggers lack in order to avoid the months of manual labour spent tracking down a single deceitfully malignant bug?

### 4.2.1 Determinism and Reproducibility

As stated earlier, cyclic debugging is based on the concept of tracking down bugs by reproduction of erroneous executions in an environment adapted for thorough investigation. This environment is provided by the debugger tool. However, for this scheme to work, it is fundamental that we are able to reproduce the program behaviour that we want to examine in the debugger environment. For regular sequential programs with no interaction with an external process,

this is not a problem. If such a program is started an arbitrary number of times with identical inputs, all invocations of the program will exhibit identical behaviours and produce identical outputs. A program with this property is said to be *deterministic*. If an execution  $E_1$  of a deterministic program  $P$  with input  $X$  leads to a failure, all subsequent executions  $E_N$ , where  $N \geq 2$ , with input  $X$  will produce the same failure. For example, consider the program in Figure 4.2. If the program is executed with an input value of 0, it will eventually come across a division by zero. If the execution is repeated with the same input, we will again end up at the division by zero. Naturally, there is no limit on the number of times this failure will occur. Each time this program is executed with the input 0, we will reach the division by zero. Intuitively, the cause of such a failure is not very hard to find, especially with the help of a debugger.

```
int avg(int x)
{
    int y, ret;
    y = getTotal();
    ret = y / x;

    return ret;
}
```

Figure 4.2: A sequential deterministic program.

Thane [29] gives a more formal description of determinism and reproducibility with respect to debugging. It is stated that a system is deterministic if its behaviour is uniquely defined by an observed set of necessary and sufficient conditions or parameters. Reproducibility, on the other hand, requires determinism *and* the ability to control the conditions and parameters that uniquely define the behaviour of the system.

Using these definitions of determinism and reproducibility, a system or program needs to be not only deterministic, but also reproducible in order to be investigated properly by cyclic debugging. Unfortunately, not many programs are even deterministic. In fact, when looking at commercial software today, very few systems exhibit a deterministic behaviour. Over the years, efforts to improve efficiency, interactivity and hardware utilization has increased overall software complexity and determinism has often drawn the shorter straw. This

path of progress has led to a situation where a massive amount of valuable software engineering time is spent searching for bugs in non-deterministic systems without the proper help of cyclic debugging debuggers [21].

Starting with the next section, we will try to give a clear picture of the problems involved in cyclic debugging of non-deterministic systems.

### 4.2.2 Context Issues

Any useful computer system interacts in some way with its environment. In its most basic form, an example of such interaction could be that of a program taking a command line input, sequentially calculating something based on that input and finishing by returning a command line output. This example system is depicted in Figure 4.3. If the command line input given to this program is the only parameter that can affect the output of this program *and* we know of this input (after all, we gave it on the command line in the first place), to us, this is a deterministic system. In addition, since we gave the input, we can simply give it again and receive the same behaviour and the same output. Thus, this program is also reproducible.

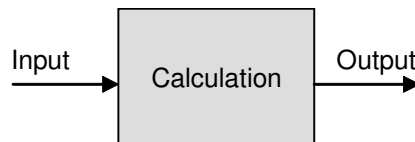


Figure 4.3: A basic interaction with a known and controllable external context.

This might seem somewhat trivial, but as we continue our line of reasoning and consider the system shown in Figure 4.4, we come across a slightly more complex situation. Suppose that the system is part of a mechatronic control system. It can be initialized and terminated by a user, but in between these events it leads a life of its own within a loop. This loop structure is used to periodically sample some external process, calculate a response to these samples and to produce actuating values based on the result of the calculation. As the observed system behaviour is uniquely defined by the inputs read at sampling time, this is a deterministic system. However, without the help of some kind of recording mechanism, it is nearly impossible to reproduce these inputs. Therefore, the system in its basic form is not reproducible and not suited for cyclic debugging.

In short: A system, whose behaviour depends on parameters provided by an outside environment, is not reproducible if we cannot control this environment. To make the system reproducible, our only option is to gain control of these parameters or the environment providing them.

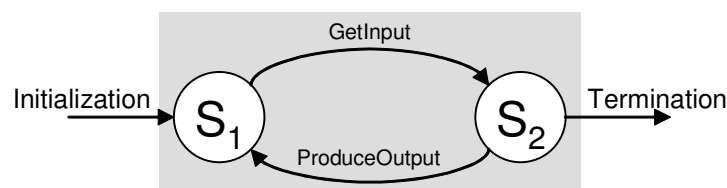


Figure 4.4: An interaction with a known, but uncontrollable external context.

### Environment Simulators

Instead of a recording mechanism that logs system inputs as they are sampled, an environment simulator could be considered for input reproduction during debugging. However, since reproducibility of a deterministic system calls for a total control of the parameters that define its behaviour, environment simulators are seldom used for reproducing exact behaviour of systems for debugging. Although some of these simulators are capable of emulating an environment fairly exact in normal situations, they often fail to reproduce those odd situations that often precede a system failure [28]. In addition, in the discrete and inherently chaotic context of software, the difference between being *fairly exact* and *downright exact* may be very significant. Therefore, environment simulators will not be further discussed in this report.

### 4.2.3 Ordering Issues and Concurrency

So far, we have only focused on sequential programs, i.e. programs running in a single thread of execution on a single processor. Looking at current state-of-the-practice commercial software, the number of applications that fulfill these restrictions is easily accounted for. In order to meet requirements regarding efficiency, interactivity and hardware utilization, most applications are pseudoparallel (several threads of executions on the same processor) or parallel (several threads of executions on several different or identical processors). If the interaction with an external context described in the above section seemed

troublesome with regard to cyclic debugging, it is merely a minor problem compared to the problems introduced by the concept of concurrency. Although debugging truly parallel systems introduces additional complexity compared to debugging of pseudoparallel systems, the main focus of this report will be on the latter. Henceforth, we will use the term *concurrent* to describe a system that is parallel or pseudoparallel.

To see why cyclic debugging of concurrent programs has the potential to slowly drive a programmer into early retirement, we consider a very central question in the process of software engineering, namely:

*What caused my program to fail?*

This question does indicate that at some point in time, prior to the failure, something happened that eventually led to the failure. In Section 4.1, we referred to this event as the *system infection*. Now, suppose this is a sequential program and it is investigated during a few iterations in a debugger. The problem has now been narrowed down and the initial question transformed into an equally central question of software engineering:

*How come the variable  $x$  is assigned the value zero?*

Apparently, the information gained during the debugging session has helped us to the conclusion that the system failed because of the fact that a variable holds an erroneous value. This new question becomes the platform from which we continue our debugging. After some further investigation, the question has transformed again:

*Why are no global variables properly initialized?*

This iterative process of debugging and narrowing down the scope of investigation continues until the bug is found and corrected. However, the actual cause of the failure is not what is important here. The point we are trying to make is that there is an ordering of events that can be followed from the system infection up until the system failure. Analogously, there is an ordering of events that starts at system start-up and leads to the system infection. In concurrent systems, however, a reproducible ordering of events is not guaranteed, if even probable. As the system will behave differently under different orderings of events, regular cyclic debugging of concurrent systems is problematic.

As an example of this, consider a preemptive priority-based multi-tasking system  $S$  with two tasks,  $A$  and  $B$ . The tasks share a common variable  $y$  and

accesses to this variable may or may not be protected by mutual exclusion constructs. In a test run of  $S$ , task  $A$  starts to execute and at time  $t_0$ , it initializes  $y$  to 0. At  $t_1$ , task  $A$  is preempted by task  $B$ , which has a higher priority. Task increments  $y$  and again passes control over to task  $A$ , which finishes its execution according to the system specifications and all is well. This scenario is shown in Figure 4.5.

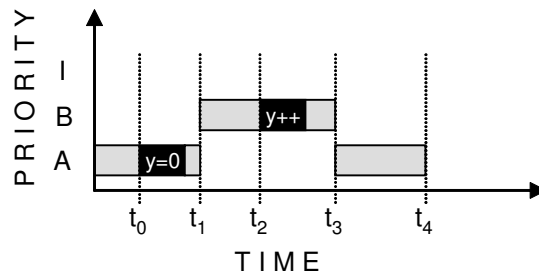


Figure 4.5: A correct ordering of events in system  $S$ .

We now consider the system  $S$  to be correct under the above circumstances. The system is deployed, but shortly after deployment, failures start to occur. Following massive investigations, it is discovered that an interrupt  $I$  occasionally temporally interacts with task  $A$  and  $B$  such that the initialization of the shared variable  $y$  is significantly postponed. This unforeseen mishap alters the ordering of events in the execution, leading to a system failure. The erroneous scenario is depicted in Figure 4.6. Since ordering factors are able to alter the behaviour of  $S$ , the system is non-deterministic with respect to input alone. For  $S$  to become reproducible, the ordering of events upon which the behaviour of the system depends must be known and controllable.

### System-Level Control Flow

The term *control flow* is used to denote the path that is taken by a thread of execution through a piece of code. The control flow contains information about branch selections, loop iterations and recursive calls visited.

Analogously, we use the term *system-level control flow* to describe the sequence of task interleavings, preemptions and interrupts in a multi-tasking system. The system-level control flow could also include invocations of semaphore

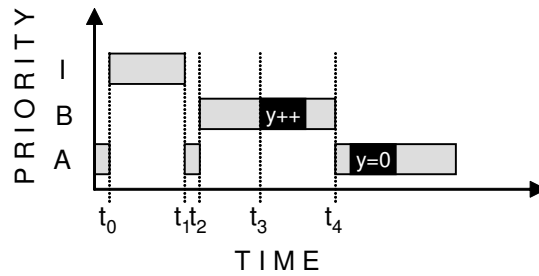


Figure 4.6: A faulty ordering of events in system  $S$ .

primitives or other synchronization mechanisms. If we were to look back on Figure 4.5, that picture describes a system-level flow of control from task  $A$  to task  $B$  and back again to task  $A$ . If the critical sections accessing  $y$  are protected by mutual exclusion constructs, the entry and exit operations of these critical sections are also included in the system-level control flow information. In other words, we use the system-level control flow to describe which task or interrupt service routine has control of the CPU at any given time, at exactly which times and locations this control is transferred to another task and why this transfer of control occurs.

The System-Level Control Flow information is important because it lets us derive the ordering of events significant to the behaviour of each individual execution. Typically (as in the above example), inaccurately synchronized and protected shared-memory accesses may result in system failures. Manifestations of such bugs are traditionally denoted *Race Conditions*. In a journal article from 1992 [20], Netzer and Miller formalize and categorize these Race Conditions. The authors differentiate between two different types of races:

- **General Races**

General races occur in improperly synchronized programs intended to be completely deterministic. Such a program could for instance be a parallelization of a sequential program, where the parallel program is forced into the same semantics as the sequential program by means of explicit synchronization. If this synchronization fails, the program will exhibit general races.



- **Data Races**

Data races describe the situation where critical-section accesses, intended to be atomic, are non-atomic due to erroneous use- or lack of mutual exclusion mechanisms in the program. In other words, a data race requires a possibility for a process or a task to execute a critical section access during the same time interval that another process or task accesses that very critical section.

Netzer and Miller also classify race conditions in another dimension. Looking at which races that actually can occur, it is possible to differentiate between feasible- and apparent races, where the apparent races are races that seem possible, looking at the semantics of the explicit synchronization constructs of a program, whereas the feasible races are the races that can actually occur during execution of that program. Thus, the set of all feasible races of a program constitute a subset of the set of all apparent races of that program.

#### 4.2.4 Timing Issues

When discussing the concept of system-level control flow, there is an important distinction to be made. System-level control flow events can be either synchronous or asynchronous. Synchronous events are manifestations of the internal synchronization semantics of a system (e.g., message receipts, semaphore releases or delay calls), whereas those events categorized as asynchronous occur due to corresponding events in the external temporal- or functional context (such as timer- or peripheral interrupts). As a consequence, synchronous events always occur at pre-determined locations or states of a program (a semaphore will always be released at a `semaphore_release` operation). In contrast, asynchronous interrupts can occur anywhere outside sections protected by `interrupt_disable` operations. This temporally and logically unrestricted access of asynchronous events makes the location of occurrence of these events inherently harder to pinpoint than that of synchronous events. Methods for handling this problem have been proposed and will be discussed in Section 4.3.3.

In 1978, Lamport presented a method for achieving a total ordering of events in a distributed system execution [15]. This paper addressed the problem of lack of a mutual timebase in multiprocessor systems. As an example, consider a two-processor system (processor *A* and *B*) with local per-processor clocks. As it is practically impossible to achieve a perfect synchronization of these clocks, they will differ with a precision of, say,  $2\delta$ . Given this, ordering

an event  $e_1$  occurring at local time  $x$  in processor  $A$  and an event  $e_2$  occurring at local time  $x + \delta$  in processor  $B$  will be very difficult. In Lamport's proposal, this problem is solved using a distributed algorithm based on per-node logical clocks rather than physical clocks. In addition, Lamport's solution allows for events on different nodes to have indistinguishable logical timestamps. Such events are said to be concurrent.

Even if we do not focus on distributed multiprocessor systems, Lamport's paper makes an important point: A correct reproduction of the temporal behavior of an execution implies a correct reproduction of the ordering of events in the execution. However, a correct reproduction of the ordering of events does not necessarily imply a correct reproduction of the temporal behavior. In a more formal notation:

*Timing*  $\Rightarrow$  *Ordering*, whereas *Ordering*  $\not\Rightarrow$  *Timing*

In many systems, a correct reproduction of the ordering of synchronous events is sufficient for achieving a correct reproduction of the entire system behavior. However, in systems with real-time requirements, such as most safety-critical systems, alterations in the temporal behavior of an execution that do not explicitly alter the ordering of events might nevertheless affect the correctness of the system. An example of a system with this type of behavior is a multi-tasking real-time system with hard task deadlines.

In addition, relying solely on the synchronization ordering for reproducing non-deterministic executions will only work if the system is properly synchronized. If data races exist in an execution, these will go by undetected if we focus exclusively on the correct reproduction of the synchronization sequence. Solutions to this problem have been proposed (discussed later in Section 4.3.4), but all proposals aim at achieving *race detection* and not *race reproduction*.

Another issue related to timing is the problem of maintaining the correlation between the internal- and the external timebase while simultaneously examining a system. As an example, consider the development of an ABS-braking system for a car. During the testing phase of the system, a failure is discovered and the system is run in a debugger. However, breaking the execution of the system by setting a breakpoint somewhere in the code will only cause the program to halt. The vehicle, naturally, will not freeze in the middle of the maneuver, hence the shared time base of the system and the external context is lost. Problems of this nature, when the intrusive examination of a system affects the system behavior itself, are denoted *Probe Effects* [10], and will be discussed later in Section 4.4.1.

### 4.2.5 Embedded Systems

In addition to the theoretical issues discussed in the above sections, there is a more practical issue related to debugging of systems that are embedded (as is the case of many real-time systems). Due to the embedded nature of such systems, the number of resources for interaction is very low, making the high interactivity needed for the debugging process troublesome to achieve. This should be considered in contrast to debugging in desktop system environments, where the machine used for debugging logically and physically is the same as the one running the program being debugged. To overcome this problem, several methods and tools have been developed. Basically, these can be divided into two categories:

- **Simulator- or Emulator-Based Embedded Debugging**

Simulator-based embedded debugging solves the problem of lacking peripheral resources by using highly interactive software target simulators or hardware target emulators instead of actual target machines during debugging sessions. Running software simulators can be either operating system-level simulators (e.g., VxWorks [14]) or hardware-level simulators (e.g., gdb [24] or IAR [9]). As shown in Figure 4.7, these simulators run as an application program on the debugger host.

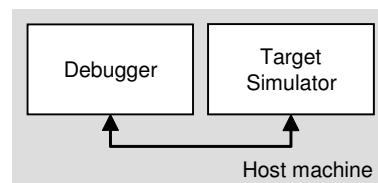


Figure 4.7: Using simulator-based debugging, the target simulator runs on the host.

Hardware- or In-Circuit Emulators (ICE:s), on the other hand, are dedicated physical machines with the task of emulating a target hardware platform, while at the same time providing an extensive interface for system investigation.

- **Remote Embedded Debugging**

The other paradigm for embedded system debugging is that of remote debugging. In this type of solution, the actual target hardware is used for

executing the investigated system during debugging sessions. All communication with the host (such as setting of breakpoints, investigation of data or single-stepping) is handled over standard communication infrastructure (such as ethernet) by a dedicated on-target debugging task, or by means of specialized on-target debugging ports. As depicted in Figure 4.8, the JTAG [25] and BDM [11] standards are examples of the latter.

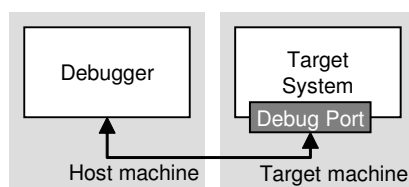


Figure 4.8: Remote debugging uses the actual target hardware for debugging.

However, in many state-of-the-practice systems, the choice of debugging target paradigm has been made transparent to the user performing the debugging, and from a host environment point-of-view, systems can be debugged similarly regardless of the logical- and physical location of the target system.

#### 4.2.6 Complex Debugging Summary and Problem Statement

To summarize this section, there are a few issues that have to be resolved when trying to apply cyclic debugging to multi-tasking real-time programs. At the start of this section, we started with a sequential, non-real-time program with a behavior depending solely on its command-line inputs. An (easily reproducible) execution of such a system is shown in Figure 4.9.

As systems are incorporated in different temporal- and environmental contexts, the assumption of a completely deterministic- and reproducible system behavior will begin to crack. Interactions with external contexts and multi-tasking will lower the probability of traditional execution reproducibility to a negligible level. As depicted in Figure 4.10, events occurring in the temporal external context will actively or passively have an impact on the internal state of the system execution. In this section, we discussed issues related to context, ordering, timing and the embedded nature of embedded systems. Now, let us see if we can derive a less abstract problem formulation from these sections.

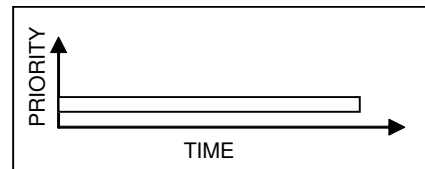


Figure 4.9: Execution of a deterministic sequential program.

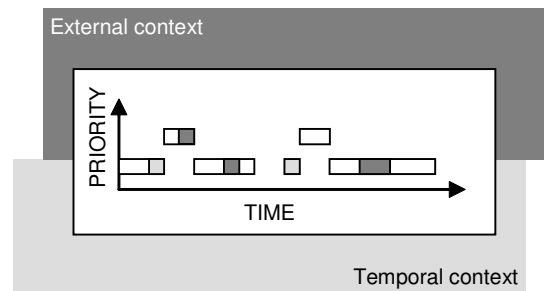


Figure 4.10: Execution of a multi-tasking real-time system, dependent on the temporal- and the external context within which it is executing.

In order to formulate our problem correctly, it is imperative that we are clear on the issue of what our goal is.

*We aim at achieving the same level of reproducibility in multi-tasking real-time systems as the one we have in non-real-time sequential software.*

Considering this, there are a number of clarifying steps we can take. First, even though they served a logical- and (hopefully) pedagogical purpose, all issues related to ordering can be ignored. This might seem odd, but consider the conclusion in Section 4.2.4. Since timing implies ordering and we need to be able to reproduce timing in order to meet our requirements, the reproduction of system timing will have correct reproduction of event ordering as a consequence.

Second, from the point of view of the temporal and external contexts, the interaction with the system manifests itself in two different ways: The con-

text (or rather a peripheral device) can actively interact with the system by means of an asynchronous interrupt. This interrupt most often has its origin in a corresponding event in the context itself (e.g., a reset of a clock register or a completion of an I/O transaction). The other means of interaction is when the context is passively being read by the system (e.g., synchronous reads of sensor values or clock registers).

Third, as the issues related to embedded systems are more of an interactive nature and do not directly relate to reproducibility, these are not considered at this stage. Hence, there are three and only three things that are required for deterministic reproduction of executions for debugging on a single-node concurrent system:

- **Starting State**

We need to be able to provide the exact initial state of the execution we seek to reproduce.

- **Input**

Any input, initial or intermediate, to the execution must be reproduced, such that it exactly simulates the temporal, external or random context of the execution we desire to reproduce.

- **Asynchronous Events**

All asynchronous events that occurred within the execution must be reproduced in such a way that their temporal and causal interference with the execution is not altered.

The first item of this list of requirements is no different to that of reproduction of non-real-time sequential software (i.e. command-line input), albeit slightly more difficult to implement in real-time multi-tasking software. This requirement is analogous to that of being able to pinpoint the current location on a map in order to use the map correctly. If you do not know where you are, there is no use reading the map in order to get where you want to be.

The second and the third requirement simply reflect the influence of temporal and external means of passive (input) or active (asynchronous events) interaction. However, note that these requirements are based on theoretical assumptions. As we shall see in the next section, due to practical reasons, some of these requirements have been ignored, underelaborated or overelaborated in previous work.

## 4.3 Debugging by Execution Replay

As the reproducibility problem is not unique to the scope of real-time systems, some work has been performed in the area of deterministic reproduction of parallel and concurrent system executions. This work has mainly focused on solutions based on *execution replay*. Generally speaking, execution replay is a set of methods for recording information of a particular system execution during run-time (this recorded information will henceforth be referred to as the *reference execution* [12]) and to use this recorded information off-line to reproduce the behavior of the reference execution in a *replay execution*. Which on-line information to record and how to record it is a subject of debate and each method has its own proposal. What is common for all execution replay methods is the possibility of cyclic debugging of otherwise non-deterministic and non-reproducible systems during the replay execution.

### 4.3.1 Replay Debugging of Concurrent Systems

Debugging by means of execution replay was first proposed in 1987 by LeBlanc and Mellor-Crummey [16]. Their method is denoted *Instant Replay* and focuses on logging the sequence of accesses to shared objects in parallel executions. As all interactions between concurrent tasks can be modeled as operations on shared objects, this log sequence can be used to reproduce the concurrent program behavior with respect to the correct ordering of interactive events between tasks. This method requires a shared object access protocol that ensures that only read operations on the shared objects can be truly concurrent. Write operations have to be serialized such that for any two writes  $W_1$  and  $W_2$ :

$$W_1 \rightarrow W_2 \vee W_2 \rightarrow W_1$$

where the symbol  $\rightarrow$  denotes the happens-before relation, as defined by Lamport [15].

Some subsequent proposals have been extensions to the work of LeBlanc and Mellor-Crummey. For instance, Audenaert and Levrouw have proposed a method for minimizing recordings [2] of shared object accesses and Chassin de Kergommeaux and Fagot included support for threads [6] in a procedural programming extension of the Instant Replay method.

In addition, some methods have been proposed that use constructs in the run-time environment of specific languages, such as Ada and Java, for execution recording and debuggable replay. The Ada replay method, proposed by Tai

et al. in 1991 [26], records the synchronization (SYN-sequence)  $S$  of a concurrent Ada programs  $P(X)$  and uses this sequence to transform  $P$  to a concurrent Ada program  $P'(X, S)$ , which produces the same output. The Java replay was proposed by Choi et al. in 2001 [7]. This method, denoted DeJaVu, uses the internal constructs of the Jalapeño virtual machine [1] in order to instrument, log and reproduce the thread interleaving sequence of cross-optimized multi-threaded Java programs. DeJaVu also claims to be able to reproduce the occurrence of asynchronous events. However, these events may only occur at pre-determined yield-points in the program, making their asynchronous nature a subject of debate.

It should be noted that none of the above actually consider the impact of real-time events, such as interrupts, on the behavior of the system. When focusing solely on a correct reproduction of the synchronization sequence of an execution, we might be able to detect bugs related to faulty synchronization (general races), but we will not be able to guarantee correct reproduction of unsynchronized accesses to shared data (data races). In addition, we cannot reproduce the occurrence of asynchronous events such as hardware interrupts. These problems have been addressed and some proposals will be discussed in Sections 4.3.3 and 4.3.4.

As for intermediate input and starting conditions, all of the above proposals assume that all necessary input and parametrization is given at the time of system startup. Hence, the starting condition of the replay executions is given by means of command-line input or similarly. Unfortunately, this makes replay of programs with a periodic interaction with an external context impossible. In addition, this might imply problems when debugging long-running programs, as discussed in Section 4.6.

Considering the list of requirements presented in Section 4.2.6, the above proposals lack support for asynchronous event- and intermediate input reproduction. This might seem a little odd, but considering the scope of these proposals, these replay methods are well-suited for the area of concurrent programs running in desktop environments. In such systems, context switches, asynchronous events and peripheral routines operate on an abstraction level far below the one of user applications. In addition, the concurrent program which we wish to reproduce might be one of a plentitude of programs running at the same time on the same machine. All these concurrent programs share resources and interact with each other in a temporal- and possibly also functional manner. Concentrating on an exact reproduction of the synchronization sequence in such an environment can be a sound design choice. However, it will leave us with a less exact reproduction of the recorded reference executions.



In contrast, when dealing with a small, embedded real-time system, the level of execution control is often higher. Dynamic spawning of tasks and allocation of memory are rarities. In addition, since most real-time systems are dedicated and designed for a single purpose, it is often desirable to perform a system-level debugging rather than a program-level debugging.

### 4.3.2 Real-Time Systems Replay Debugging

In the area of execution replay for real-time systems debugging, results have been very scarce and all contributions known to us apart from our work, are at least ten years old. In 1989, Banda and Volz [4] proposed a method for reproducing the execution behavior of real-time systems using a non-intrusive hardware-based recording and replay mechanism. In addition to the dedicated monitoring hardware required, this method also called for specialized compiler support. Similarly, Tsai et al. proposed a non-intrusive hardware-based monitoring- and replay mechanism [33], craving a highly specialized hardware platform. Tsai's method has been criticised for an overelaborate logging scheme.

In contrast, Dodd and Ravishankar [8] proposed a software-based replay method for the distributed multiprocessor-node real-time system HARTS [23]. The logging is software-based in that it is performed by intrusive software probes. However, in order for the method to work, a dedicated processor needs to handle the monitor processing on each node.

### 4.3.3 Asynchronous Events Reproduction

Since very few general replay methods support reproduction of asynchronous events, a number of specialized tools and methods for this purpose have been proposed. When trying to replay executions containing elements of asynchronous nature, it is not only the reproduction of such elements that is difficult, but also the task of determining their exact location of occurrence.

For example, consider the execution in Figure 4.11. Two tasks,  $A$  and  $B$  share a mutual resource and this resource is accessed within critical sections, protected by semaphores in the code of each task (marked black in the figure). In our example, a system clock interrupt at time  $t_0$  invokes the task scheduler and causes task  $A$  to be preempted by task  $B$ . The latter enters its critical section at  $t_1$  and manipulates the mutual resource. The semaphore is released and at time  $t_2$ , task  $B$  lets task  $A$  resume its execution. At  $t_3$ , task  $A$  enters its critical section and accesses the mutual resource, already manipulated by  $B$ .

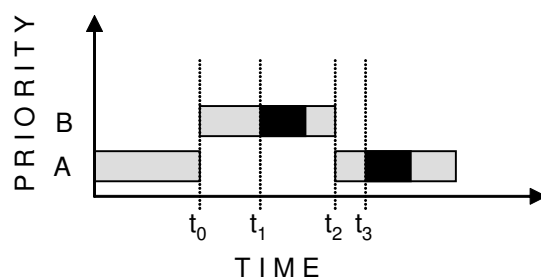


Figure 4.11: Faulty execution due to erroneous event ordering.

Now, assume that this ordering of events leads to a violation of the system specifications, i.e. a failure. We start up the debugger and try to reproduce the error in an environment more suited for thorough system investigation. However, our inability to reproduce the clock interrupt at the exact same program state as in the first execution leads to a different outcome in the race for the mutual resource. As we can see in Figure 4.12, task *A* is now able to acquire the semaphore before task *B* is scheduled for execution. Depending on the protocol used for semaphores in the system, this might give rise to several different scenarios. However, we assume semaphores of a basic mutual exclusion implementation.

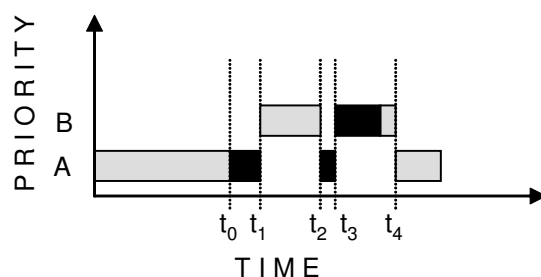


Figure 4.12: Correct execution and event ordering.

At time  $t_1$ , the clock interrupt hits the system and task  $B$  is scheduled for execution. Since task  $B$  has a higher priority than task  $A$ , the latter is preempted and  $B$  is executed up until time  $t_2$ , when it tries to grab the semaphore and enter its critical section. This time, the semaphore is already taken and task  $A$  is resumed. At  $t_3$ ,  $A$  releases the semaphore and  $B$  is allowed to enter its critical section, manipulate the resource and leave the critical section before task  $A$  is allowed to finish at time  $t_4$ . As a consequence, the outcome of this ordering of events does not violate the system specification.

#### Pinpointing the Location of Occurrence of Asynchronous Events

In the above example, the source of the inability to reach the identical sequence of system infection and error propagation is the inability to reproduce the interrupts of the reference execution in a deterministic fashion. In other words, the asynchronous events of the execution cannot be reproduced in such detail that the erroneous states that they caused are revisited in the replay execution.

The problem of monitoring asynchronous events is the difficulty of pinpointing their exact location of occurrence. As we saw in the above example, a correct reproduction of the occurrence of asynchronous events is often an absolute necessity for correct reproduction of the overall reference execution.

As another example of this problem, consider the code in Figure 4.13. When executed, an interrupt occurs at program counter value 0x8fac, leading to a task switch. Since this interrupt is an asynchronous event, it needs to be recorded. We can easily store the time of the event (although this time is too inexact to serve as an indicator of the location of occurrence of the event since the number of instructions executed during a fixed time interval may vary [30]), and the program counter value at which it occurred. It might seem feasible to reproduce the preemption at program counter value 0x8fac during the replay execution and thus solving the problem of pinpointing the location of the event. However, since the interrupt occurred within a loop, this program counter value might be revisited a number of times. There is no way of telling during which iteration of the loop the interrupt occurred.

To solve this problem, the program counter location marker needs to be extended with a unique marker, helping to differentiate between loop iterations, subroutine calls and recursive calls. The content of this marker should be chosen such that it uniquely defines the state of the program at the occurrence of the event.

```
subroutine_A() {  
    int i, a = 0;  
    rdSensor(s1, &a);  
    for(i=0; i<10; i++) {  
        smtUseful1(a, i);  
        smtUseful2(a, i);  
    }  
}
```

PC = 0x8fac

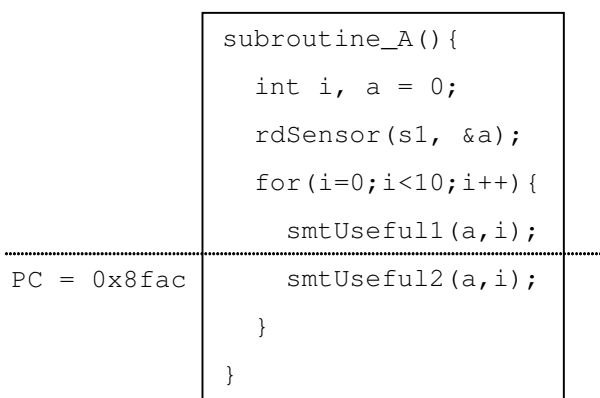


Figure 4.13: Interrupt occurs while in a loop.

### Unique Marker Techniques

Previous work aimed at pinpointing the location of occurrence of asynchronous events has focused on using different types of instruction counters (IC) as unique markers. Basically, an instruction counter in its original form is a counter, incremented at the execution of each instruction. Since such a counter is infeasible to implement in software (it would require an additional incrementing instruction to be executed after each instruction of the application machine code), traditional instruction counters are a hardware-based feature of some processor platforms [5][13]. To use the value of the instruction counter as a unique marker, it is sampled at the occurrence of the asynchronous event. Since the instruction counter differentiates between executions of single instructions, it can be used to differentiate between arbitrary program states.

Practical as they may seem, hardware-based instruction counter markers are not without drawbacks. Firstly, very few (if any) operating systems, real-time or regular, support the sampling of the counter value at the occurrence of asynchronous events, making the method difficult to apply to any platforms using operating systems. Secondly, while some desktop-level processors, such as the Intel Pentium and the PowerPC, feature instruction counter support, most embedded processors do not. Without this hardware support, embedded processors need another means of pinpointing asynchronous events. Therefore, in

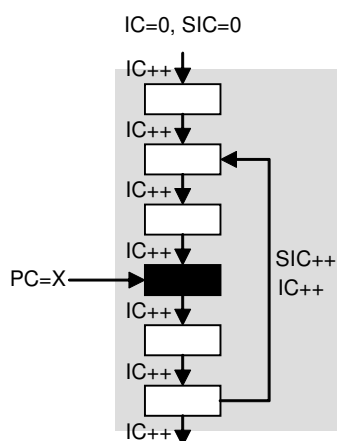


Figure 4.14: Incrementations of hardware- and software-based instruction counters.

1989, Mellor-Crummey and LeBlanc proposed the usage of a software-based instruction counter (SIC) [17]. To avoid the problem of massive software instrumentation mentioned above, the SIC is more selective upon which instructions to increment. As a traditional instruction counter increments at *every* instruction executed, the SIC only increments at branches and subroutine calls. Since these instructions are the only ones that can cause program counter values to be revisited, the SIC together with the PC value and the original hardware-based instruction counters serve equally well as unique markers for asynchronous events. For example, consider the program structure in Figure 4.14. Suppose each square represents an instruction and the vertical arrows represent the sequential order of execution. In addition, suppose the square with two outgoing arrows represents a conditional branch instruction and the black square represents the location of occurrence of an asynchronous event. As a traditional instruction counter would increment at every instruction, the event might have occurred at an IC value of 4, 9, 14, 19 and so on. In contrast, the SIC only increments at the backward branch and a SIC value of 0 together with a PC value of X pinpoints the exact same location as an IC value of 4. Analogously, a SIC value of 1 and a PC value of X is equivalent to an IC value of 9.

Using software instruction counters, there is no need for specialized hardware when pinpointing the location of occurrence of asynchronous events. Hence, the method is more platform-independent than the traditional instruction counter. However, the SIC consumes approximately 10% of the overall CPU utilization. In addition, it requires support from the compiler in the form of a dedicated SIC processor register and from the operating system for sampling at the occurrence of events. It also requires a tool for machine code instrumentation that is target specific.

The instruction counter and software instruction counter techniques were proposed as independent techniques, unbiased to any previously proposed replay method. However, in 1994, Audenaert and Levrouw proposed the use of an approximate software instruction counter [3]. The proposal was part of an extension of Instant Replay, denoted Interrupt Replay, which added support for interrupts to the Instant Replay method. In Interrupt Replay, the run-time logging of system interrupts is done by recording interrupt ID and an approximate SIC value. This version of SIC is incremented at entry- and exit points of interrupt service routines and can therefore not be used to pinpoint exact locations (program states) of occurrence of interrupts. As a consequence, Interrupt Replay is able to replay orderings of event sequences, but not with the correct timing. In addition, the technique is dependent on the absence of interrupt races in the system, meaning that interrupts to be replayed can not access data used by other running threads or processes, as this invalidates correct reproduction of the execution.

#### 4.3.4 On-The-Fly Race Detection

In Section 4.3.1, we stated that most previous work in the area of concurrent systems replay debugging require programs with an explicit synchronization of all shared memory accesses. Since this is a highly limiting restriction, methods have been proposed to handle unsynchronized accesses to shared memory (data races) during program execution. These methods are known as *On-The-Fly Race Detection* methods and their basic idea is to search a concurrent program for data races as it is running. As race detection is not the main focus of this paper, we will not go into detail of how this works. However, it is of importance that we differ between two types of race detection:

- **Detection of Apparent Data Races**

Detection of apparent data races is not exact in that it will detect and report all data races, even though they are not feasible.

- **Detection of Feasible Data Races**

Detection of only feasible data races is exact, since it only detects and reports races that can actually occur. It is, however, a far more complex and time-consuming task than the apparent race detection.

As a consequence, when using apparent race detection methods, an execution could be reported to include hundreds of data races even though none of these could actually occur. In the worst case, this could lead to large amounts of time spent correcting bugs that do not really exist. Unfortunately, the more correct feasible race detection methods can not be used during traditional run-time, since the complexity of the race detection algorithm is too high. However, combining on-the-fly feasible race detection with a replay method, able to reproduce the correct synchronization ordering of a concurrent program, we can work around the complexity issue. As the correct ordering of synchronization events already is recorded, the on-the-fly race detection algorithm can safely be run during the replay execution. As a data race is detected, the replay is stopped and subsequent determinism in the execution can not be guaranteed. Some even argue that replay combined with automatic on-the-fly race detection is preferable when compared to a “passive” deterministic replay, which correctly reproduces data races instead of detecting and reporting them. It should be noted, however, that a replay/on-the-fly race detection method does not provide the real-time properties needed for correct reproduction of events of asynchronous nature.

An example of a race detection method was proposed by Ronsse and De Bosschere [22] in 1999. Their method, RecPlay, is implemented in the form of a hybrid replay/on-the-fly race detection tool. During run-time, RecPlay collects the synchronization sequence of a concurrent Solaris program. Then, during replay it uses an on-the-fly data race detection method to find data races in the replay execution. If a race is found, the execution stops and the remainder of the replay is invalidated.

## 4.4 Instrumentation for Replay

All replay methods require some form of recording of important system events during the reference execution in order to be able to reproduce the system behavior during the replay execution. This system monitoring and recording can be performed by means of software-, hardware- or hybrid mechanisms [32]. What type of information that should be recorded is individual for each replay method. However, most methods require information of synchronization

races and input data, which calls for an instrumentation of the synchronization layer (e.g., message passing- and semaphore primitives) and the I/O layer of the system.

Intuitively, software-based instrumentation performed by software-based probes incorporated in the system will consume a certain amount of the overall CPU time. Software-based instrumentation is therefore sometimes denoted *intrusive* instrumentation, whereas hardware-based instrumentation is denoted *non-intrusive* instrumentation. Apart from the performance drawbacks, there are a few issues related to intrusive software- and hybrid instrumentation. These issues are discussed in the following sections.

#### 4.4.1 The Probe Effect

As software-based probes are intrusive with respect to the resources of the system being instrumented, the very act of observing may alter the behavior of the observed system. Such effects on system behavior are denoted *probe effects* and their presence in concurrent software were first described by Gait [10].

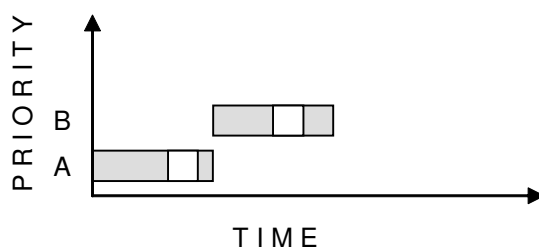


Figure 4.15: An execution leading to a system failure.

The cause of probe effects are best described by an example. Consider the two-task system in Figure 4.15. The two tasks (*A* and *B*) share a resource, *X*, accessed within critical sections. In our figure, accesses to these sections are displayed as white sections within the execution of each task. Now, assume that the intended order of the accesses to *X* is *first* access by *B*, *then* access by *A*. As we can see from Figure 4.15, the intended ordering is not met and this leads to a system failure.



Since the programmer is confused over the faulty system behavior, a probe (e.g., a `printf`-statement), represented by the black section in Figure 4.16, is inserted in the program before it is restarted. This time, however, the execution time of the probe will prolong the execution of task *A* such that it is preempted by task *B* before it enters the critical section accessing *X* and the failure is not repeated. Thus, simply by probing the system, the programmer has altered the outcome of the execution (s)he wishes to observe such that the observed behavior is no longer valid with respect to the erroneous reference execution. Conversely, the addition of a probe may lead to a system failure that would otherwise not occur.

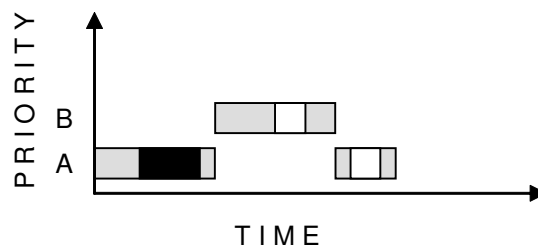


Figure 4.16: The same execution, now with an inserted software probe, “invalidating” the failure.

In concurrent systems, the effects of setting breakpoints, that may stop one thread of execution from executing while allowing all others to continue their execution, thereby invalidating system event orderings, are also probe effects. The same goes for replay instrumentation. If the system probing code is altered or removed in between the reference- and the replay execution, this may manifest in the form of probe effects.

#### 4.4.2 Instrumentation Jitter

Apart from the probe effect, when using software- or hybrid-based instrumentation in real-time systems, there is a less intuitive, but slightly related, problem to be considered. Real-time systems, especially hard real-time systems, are often temporally well-designed with well-defined behaviors. The execution behavior analysis needed to achieve these properties is made easier by

minimizing jitter in the system. Jitter is the term we use to denote execution time variations for different parts of the system. For example, depending on the number of active tasks, the state of each task, synchronization- and message mechanisms, the time spent in the kernel task scheduling routine might differ. These temporal variations are part of the kernel jitter.

As the jitter in the system grows, the number of possible execution paths grows exponentially [31], making the system harder to analyze and test. Being software-based, intrusive probes may also exhibit variations in execution time due to different branch selections. This could lead to the somewhat strange situation where mechanisms inserted with the intention to increase the ability to debug the system have the side-effect of reducing the testability of the system. In addition, even though software probes may not be part of the actual system, they might interact with the system in a temporal manner. Therefore, replay methods must make sure that jitter in the instrumentation during the replay execution does not compromise the deterministic reproduction of the reference execution.

## 4.5 Deterministic Replay

In Section 4.3, we stated that very little work was done in the area of real-time systems replay debugging. However, in 2000, just after a decade of dead calm in this field of research, Thane and Hansson proposed a software-based approach to distributed real-time systems replay debugging, denoted *Deterministic Replay* [30]. In many respects, the Deterministic Replay method is similar to previously proposed methods for concurrent system replay debugging, such as Instant Replay, but in some significant respects, it differs. First, Deterministic Replay, as proposed by Thane and Hansson, is an integrated solution, craving instrumentation support from a specialized real-time kernel. This support enables both synchronous- and asynchronous events affecting system behavior to be monitored. Interrupts, synchronization primitive calls and task interleavings are logged during a reference execution and reproduced during a replay execution. The possibility of deterministically reproducing asynchronous events not only guarantees the correct ordering of events, but also correct timing. Second, as the method allows for input-, global- and static data to be recorded during the reference execution, this method allows for interaction with external context to be replayed. Third, as the replay technology of Deterministic Replay does not use any specialized hardware, development environment or language support, standard debuggers can be used to perform the replay execution.

### 4.5.1 Reproducing System-Level Control Flow

To be able to reproduce the system-level control flow, the Deterministic Replay method makes use of a small software probe in the kernel task switch- or interrupt routine. This probe extracts information for each task switch as it occurs, such as task id, type of control flow event, a timestamp and a program counter value together with a unique marker if the task is preempted by an asynchronous event.

In short, the information gathered by these probes is uploaded to a development host in the case of a system failure and analysed by a host-based application. The replay tool then sets breakpoints at all locations in the code where task interleavings occurred during the reference execution and restarts the systems in order to initiate the replay execution. As breakpoints are hit, the unique marker value is analysed to determine if the correct location of the next interleaving has been reached. If so, an interrupt or a task preemption is simulated by slight modifications of internal kernel structures. Then, the replay execution is resumed.

### 4.5.2 Reproducing External- and Internal Data Flow

In addition to the kernel-level control flow instrumentation, the system is instrumented by application-level software probe macros, inserted at carefully selected locations in the application source code. The task of these probes is to reproduce the interaction with the external context, such as readings of sensors, and the internal static task data, such as structures maintaining state over different iterations of control loops.

During the reference execution, these macros are used to monitor and log the data, whereas during the replay execution, the macros make sure that the correct data is fed to the system at the correct time.

## 4.6 Replaying Long-Running Applications

When we listed our requirements on a real-time replay method way back in Section 4.2.6, one of the three basic imperatives was the ability to reproduce the correct starting state of the execution to be executed. We have also stated earlier that this is no problem when dealing with simple sequential, command-line programs. If the same command-line input is given to such a program twice, both executions will behave identically and produce the same output. In other words, both executions will start at the same state and follow identical paths

through the program. In fact, this not only goes for sequential programs, since all programs, concurrent or not, have an initialization phase that is deterministic up until a certain point.

#### 4.6.1 Starting a replay execution

The real problem occurs when trying to replay long-running applications. Many of today's embedded systems have up-times spanning weeks, months or even years. If we encounter a system failure after such an execution, replaying the execution would be infeasible for several reasons. For instance, consider the long-running application in Figure 4.17. At time  $t_{fail}$ , a failure occurs. When trying to debug this system by execution replay, due to limited memory for system recording or unbearably long debugging sessions, it might be impossible to reconstruct the entire execution from time  $t_0$ . Suppose that a reproductive replay execution is only feasible from time  $t_1$  up until  $t_{fail}$ . We are then left with the problem of starting the replay execution by first creating the global, as well as task-local, system state of  $t_1$  in the reference execution.

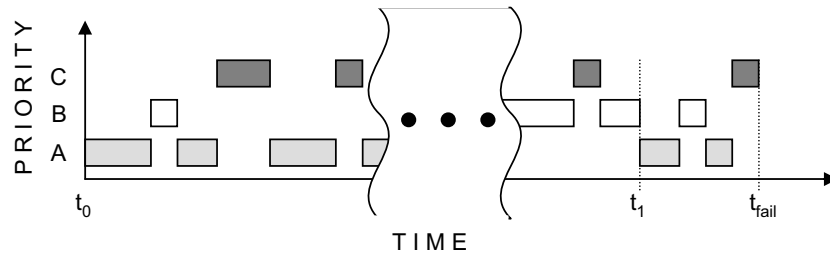


Figure 4.17: A long-running reference execution.

Unfortunately, contributions related to replay of long-running executions have been nearly nonexistent. Netzer [18] discusses the problem and uses the term *incremental replay* for denoting replay executions started at another point than system start-up. However, Netzer's proposal only focuses on reproduction of message communication in a concurrent message-passing system.

#### 4.6.2 Checkpointing

Even though our scope of systems of interest lack methods for constructing starting states different than the one of system start-up, it does not mean that it

cannot be done. When leaving the rather strictly restricted area of real-time- or concurrent system replay, methods closely related to what we wish to accomplish start to emerge. Especially in systems handling large amounts of data, such as large simulations, checkpointing is a well-known method for checkpointing the state of the system at a specific point in time [19]. What we will end up with is a snapshot of the system data area from the very instant of the checkpoint.

Checkpoints can be taken periodically as a security measure, allowing for executions to be resumed from a previous checkpoint in case of a system failure. Note that checkpointing is not a replay technique, but a method for reproducing individual system states.

## 4.7 Related Research Projects

Currently, a few groups are dedicated to research within the area of replay debugging. At Universiteit Gent in Belgium, a research group led by Professor Koen DeBosschere focuses on tools and methods related to debugging of parallel programs [22]. Results and publications have covered instrumentation, replay techniques and replay run-time automatic detection of data races. The main interest of this group has been replay debugging of parallel programs, e.g. multithreaded Java applications, rather than debugging of real-time- or embedded systems.

Another project focused on replay debugging of multithreaded Java is the DeJaVu-project [1][7], run by IBM (J.-D. Choi, B. Alpern, A. Loginov and H.-G. Yook) at the T. J. Watson Research Centre in New York. DeJaVu is a replay method that replays the entire execution of the Jalapeno virtual machine, making it possible to analyse and debug the run-time execution behaviour of multithreaded Java applications.

A more hardware-oriented research towards replay debugging is conducted at the University of Scranton by Dr. Yaodong Bi. This project suggests using specialised hardware in order to instrument and replay the execution of embedded real-time systems [33]. A small, non-intrusive monitoring processor instruments the run-time behaviour of a real-time processor. As the real-time processor reaches a failure, the monitoring processor can be used to replay the execution of the real-time system. In addition to this, a visualization method has been integrated with the monitoring and the replay tool in order to increase the possibilities of execution behaviour analysis.

## 4.8 Conclusions

In this paper, we have described the state of the art in embedded real-time systems replay debugging. We have identified the main problem with real-time system cyclic debugging (as well as debugging of other non-deterministic systems) as that of execution behavior reproducibility. Correct reproduction of erroneous execution behavior is a fundamental prerequisite for cyclic debugging methods. As most embedded real-time systems are inherently non-deterministic, correct execution behavior reproduction can not be guaranteed during debugging.

Replay debugging is a general term for a set of methods, designed to record the execution behavior of non-deterministic systems and to use these recordings in order to reproduce the execution behavior during debugging sessions. We have discussed many of these methods, including those aimed at non-real-time and real-time system debugging. Furthermore, we described the Deterministic Replay method in greater detail, since this method will serve as a foundation for the remainder of this thesis.

# Bibliography

- [1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russel, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1), 2000.
- [2] K. Audenaert and L. Levrouw. Reducing the space requirements of instant replay. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 205 – 207. ACM, May 1993.
- [3] K. Audenaert and L. Levrouw. Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts. *Journal of Microprocessors and Microsystems, Elsevier*, 18(10):601 – 612, December 1994.
- [4] V.P. Banda and R.A. Volz. Architectural support for debugging and monitoring real-time software. In *Proceedings of Euromicro Workshop on Real Time*, pages 200 – 210. IEEE, June 1989.
- [5] T.A. Cargill and B.N. Locanthi. Cheap Hardware Support for Software Debugging and Profiling. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82 – 83, October 1987.
- [6] J. Chassin de Kergommeaux and A. Fagot. Execution Replay of Parallel Procedural Programs. *Journal of Systems Architecture*, 46(10):835 – 849, 2000.
- [7] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of 15th Parallel and Distributed Processing Symposium*, page 10, April 2001.

- 
- [8] P. Dodd and C. V. Ravishankar. Monitoring and Debugging Distributed Real-Time Programs. *Software – Practice and Experience*, 22(10):863 – 877, October 1992.
- [9] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1999.
- [10] J. Gait. A Probe Effect in Concurrent Programs. *Software – Practice and Experience*, 16(3):225 – 233, March 1986.
- [11] S. Howard. A Background Debugging Mode Driver Package for Modular Microcontroller. Semiconductor Application Note AN1230/D, Motorola Inc., 1996.
- [12] J. Huselius. A Constant Queue Eviction Scheduler. Technical Report 87, Mälardalen University, Department of Computer Science and Engineering, December 2002.
- [13] M. Johnson. Some Requirements for Architectural Support of Debugging. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 140 – 148. ACM, March 1982.
- [14] A. Kornecki, J. Zalewski, and D. Eyassu. Learning Real-Time Programming Concepts through VxWorks Lab Experiments. In *Proceedings of 13th SEE&T Conference*, pages 294 – 301, March 2000.
- [15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558 – 565, July 1978.
- [16] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471 – 482, April 1987.
- [17] J. Mellor-Crummey and T. LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86. ACM, April 1989.



- [18] R. H. Netzer and M. H. Weaver. Optimal Tracing and Incremental Re-execution for Debugging Long-Running Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 313 – 325. ACM SIGPLAN, June 1994.
- [19] R. H. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165 – 169, February 1995.
- [20] R. H. B. Netzer and Miller B. P. What Are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74 – 88, March 1992.
- [21] NIST Report. The economic impacts of inadequate infrastructure for software testing., May 2002.
- [22] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133 – 152, May 1999.
- [23] K. G. Shin. HARTS: A Distributed Real-Time Architecture. *IEEE Computer*, 24:25 – 35, May 1991.
- [24] R. M. Stallman and R. H. Pesch. *Debugging with GDB; The GNU Source-Level Debugger*. Free Software Foundation, 545 Tech Square, Cambridge, Ma. 02139, 4.17 edition, February 1999.
- [25] IEEE Std. IEEE Standard Test Access Port and Boundary-Scan Architecture. Technical Report 1532-2001, IEEE, 2001.
- [26] K.-C. Tai, R.H. Carver, and E.E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, 17(1):45 – 63, January 1991.
- [27] R. N. Taylor and L. J. Osterweil. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, pages 265 – 278, May 1980.
- [28] M. Telles and Y. Hsieh. *The Science of Debugging*. The Corolis Group, 2001.
- [29] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, KTH, April 2000.

- [30] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th EUROMI-CRO Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.
- [31] H. Thane and H. Hansson. Testing Distributed Real-Time Systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24:463 – 478, February 2001.
- [32] H. Thane and D. Sundmark. Debugging Using Time Machines: replay your embedded system’s history. In *Proceedings of the Real-Time & Embedded Computing Conference*, page Kap 22, November 2001.
- [33] J.P.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transactions on Software Engineering*, 16(8):897 – 916, August 1990.

## **Chapter 5**

# **Paper B: Replay Debugging of Real-Time Systems Using Time Machines**

Henrik Thane, Daniel Sundmark, Joel Huselius and Anders Pettersson  
In Proceedings of the Parallel and Distributed Systems: Testing and Debugging  
(PADTAD) Workshop, Nice, France, April 2003.

## **Abstract**

In this paper we present a new approach to deterministic replay using standard components. Our method facilitates cyclic debugging of real-time systems with industry standard real-time operating systems using industry standard debuggers. The method is based on a number of new techniques: A new marker for deterministic differentiation between loop iterations (as well as subroutine calls) for deterministic reproduction of interrupts and task preemptions, an algorithm for finding well-defined starting points of replay sessions, as well as a technique for using conditional breakpoints in standard debuggers to replay the target system. We also propose and discuss different methods for deterministic monitoring, and provide benchmarking results from an industrial case study demonstrating the feasibility of our method. Previously published solutions to the problem of debugging real-time systems have been based on the concept of deterministic replay: where significant system events like task-switches of multitasking software and external inputs are recorded during run-time, and later replayed (re-executed) off-line. Previous works have been based on either non-standard hardware, specially designed compilers or modified real-time operating systems. The reliance on non-standard components has limited the success of the approach. Even though this idea has been around for 20 years, no industrial application for debugging of real-time systems of the method has been presented.

## 5.1 Introduction

Testing is the process of revealing failures by exploring the runtime behavior of the system for violations of the specifications. Debugging, on the other hand, is concerned with revealing the errors that cause the failures. The execution of an error infects the state of the system, and finally the infected state propagates to output. The process of debugging is thus to follow the trace of the failure back to the error. In order to reveal the error it is imperative that we can reproduce the failure repeatedly, wherefore knowledge of an initial start condition as well as a deterministic execution from the initial state to the failure is required. For sequential software with no real-time requirements it is sufficient to apply the same input and the same internal state in order to reproduce a failure. For real-time software the situation gets more complicated due to timing and ordering issues. There are several problems to be solved in moving from debugging of sequential programs (as handled by standard commercial debuggers) to debugging of multitasking real-time programs. We will briefly discuss the main issues by making the transition in two steps.

### 5.1.1 Debugging Sequential Real-Time Programs

Moving from single-tasking non-real-time programs to single-tasking real-time programs adds the concept of interaction with, and dependency of, an external context. The system can be equipped with sensors sampling the external context, and actuators interacting with the context. In addition, the system is equipped with a real-time clock, giving the external and the internal process a shared time base. If we try to debug such a program, we will encounter two major problems: First, how do we reproduce the readings of sensors from in the first run? These readings need to be reproduced in order not to violate the requirement of having exactly the same inputs to the system. Second, how do we keep the shared time base intact? During the debug phase, the developer needs to be able to set breakpoints and single-step through the execution. However, breaking the execution will only break the progress of the internal execution while the external process will continue. Consider, for instance, an ABS-breaking system in a car. During the testing phase, a failure is discovered and the system is run in a debugger. While the system is run in the debugger, the testing crew tries to reproduce the erroneous state by maneuvering the vehicle in the same way as in the first run. However, breaking the execution of the system by setting a breakpoint somewhere in the code will only cause the program to halt. The vehicle, naturally, will not freeze in the middle of

the maneuver and the shared time base of the internal and external system is lost. This makes it impossible to reproduce the failure deterministically while simultaneously examine the state of the system.

### 5.1.2 Debugging Multi-Tasking Real-Time Programs

In moving from debugging sequential real-time programs to debugging multitasking real-time programs executing on a single processor the problem of concurrency surfaces. When the system consists of a set of tasks, the tasks will interact with each other both in a temporal and the functional domain. Scheduling-events and hardware interrupts change the flow of control between tasks in the system. In addition, the sharing of resources between tasks leads to race conditions usually arbitrated by synchronization mechanisms.

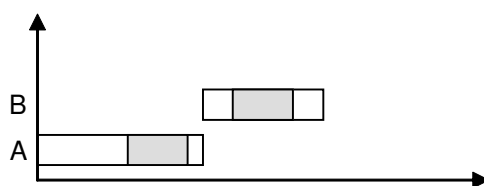


Figure 5.1: Task A executes and enters a critical section (grey area) prior to the preemption by task B. This leads to a violation of the system specification

Furthermore, as can be seen in Figures 5.1 and 5.2, the insertion and removal of probes can cause non-deterministic races. This is known as the probe effect [4][8], where the actual act of observation changes the behavior of the subject of study.

### 5.1.3 Debugging by the Use of Time Machines

We will in this paper present a debugging technique based on deterministic replay [2][7][12][13], which we call the Time Machine. During runtime, information is recorded with respect to interrupts, task-switches, timing, and data. The system behavior can then be deterministically re-executed off-line using the recorded history, and inspected to a level of detail, which vastly surpasses what has been recorded since the system is deterministically re-executed such that all calculated data is restored. We will show how entire run-time executions including: interrupts, task-switches and data can be reproduced off-line.

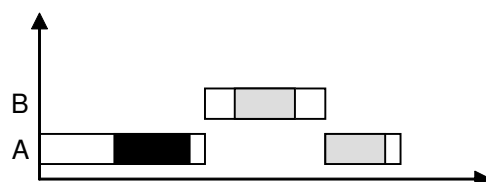


Figure 5.2: A probe is added (black area) to investigate what went wrong. The execution of the probe prolongs the execution of task A such that the critical section (correctly) is first reached by task B. Thus, the error cannot be reproduced due to the probe effect.

We will show how the system can be debugged both forward and backwards in time using standard debuggers like CPU instruction level simulators, JTAG [11] debuggers, BDM [5] debuggers or In Circuit Emulators (ICE) debuggers, with a timing precision corresponding to the exact machine code instructions at which the events occurred.

#### 5.1.4 Contribution

The contributions of this paper are: A method for debugging real-time systems, which to our knowledge is

- *The first method for deterministic replay of single tasking and multi-tasking real-time systems using standard off-the-shelf debuggers and real-time operating systems.*
- *The first to realize replay without instruction counters, using a novel approach, which is compiler and real-time operating system independent.*

Benchmarking results from the monitoring process in a recent industrial case study shows that our method is feasible to use in industry today.

Paper outline: Section 5.2 provides an overview of related work Section 5.3 presents the system model and section 5.4 the method for deterministic real-time systems debugging. Benchmarking results are presented in Section 5.5. Finally, in Sections 5.7 and 5.6 we conclude and give some hints on future work.

## 5.2 Related Work

With respect to related work in the field of replay debugging of concurrent programs and real-time systems most references are quite old and the advancement in the field has been meager. Work previously published has either been relying on special hardware [15][2], or on special compilers generating dedicated instrumented code [10][2], which has limited the applicability of their solutions on standard hardware and standard real-time operating system software. Other approaches do not rely on special compilers or hardware but lack in the respect that they can only replay concurrent program execution events like rendezvous, but not real-time specific events like preemptions, asynchronous interrupts or mutual exclusion operations [12][1][16]. For a more elaborate discussion on related work see [6]. An early version of our deterministic replay technique, which supported replay of interrupts, preemption of tasks and distributed transactions, has been presented previously [13]. However, this work assumed the existence of special off-line versions of real-time operating systems (RTOS), which is not a plausible assumption for current commercial real-time operating systems.

## 5.3 The System Model

The system software consists of a Real-Time Operating System (RTOS) and a set of concurrent tasks and interrupt routines, communicating by message passing or via shared memory. Tasks and interrupts may have functional and temporal side effects due to preemption, blocking, message passing and shared memory. We allow the execution strategy to range from interrupt driven single program systems to multi-threaded systems with real-time kernels that support preemptive on-line scheduling. We adopt the following non-terminating task models for RTOSs:

- Task activation is initiated by other application tasks or the RTOS.
- One stack for each task.
- One entry and one exit from task execution. However, the execution is contained within a while(forever) loop, or similar.
- Multiple references within task bodies to system calls which can potentially block task execution.



Of the different events that occur in the execution of the system, the occurrence of some are implied by the task bodies (e.g. system- and function calls) we label these synchronous events. The occurrence of other, asynchronous events (e.g. preemptions and interrupts), are not implied by the static code. Thus, the recording of asynchronous events must be performed more thoroughly. A subset of the synchronous events may potentially block the execution of the task, and are therefore labeled as blocking system calls (e.g. the receiving of a inter-task message). We further assume that we have access to either instruction level simulator debuggers, JTAG debuggers [11], BDM debuggers [5] or In Circuit Emulator (ICE) debuggers [9]. We assume that the debuggers have interfaces or scripting languages such that macros or programs can be invoked conditionally at specified breakpoints, as well as access to target memory. We assume for non-ICE based systems that the RTOSs have necessary “hooks” such that task switches can be recorded during runtime (most commercial RTOSs do).

## 5.4 The Mechanisms of the Time Machine

We will now in further detail discuss and describe our method for achieving time travel and deterministic replay. The basic elements of the Time Machine debugging method are:

1. **The Recorder**, which is an in-target mechanism that collects all the necessary information regarding task-switches, interrupts, and data.
2. **The Historian**, which is the off-target system that automatically analyzes, and correlates events and data in the recording, and composes these into a chronological timeline of breakpoints and predicates.
3. **The Time Traveler**, which interacts with the debugger and, given the information provided by the historian, allows the recreation of the program state (i.e. state variables, global variables, program counter, etc.) for any given time in the scope of the memory of the historian.

This process is performed without ever changing the target executable code. The same code (including RTOS) that is run in the target, during runtime, is run during the replayed execution in the debugger.

### 5.4.1 What to Record

Assuming that significant variables, like state variables, and peripheral inputs like readings of sensor values or events like accesses to the local clock, are identified and recorded for the application, it is possible to reproduce these off-line. Decoupling of the external system (the real-world) and the progression of the system is accomplished, a necessity when performing deterministic replay of real-time systems. We label the process of recording application dependent data as data-flow monitoring or -recording. Worth noting is that we need only record external inputs and internal state variables since we later during replay re-execute the system and consequently recalculate all intermediate variable values and outputs.

```
for (i=0; i<10; i++)
{
  a = a + i;
  ----- PC=0x2340
  b = q*2 + i;
}
```

Figure 5.3: The PC is not sufficient as a unique marker.

### Multitasking

To replay and debug multitasking real-time systems we need to monitor, in addition to the data-flow, the system control-flow. Essentially, the control-flow corresponds to a list of occurred task switches and interrupts, i.e., all transfers of control from one task to another task, or from a task to an interrupt service routine, and back. To identify these events, we record where and when they occur by using timestamps and the program counter (PC). However, since PC values can be revisited in loops (Figure 5.3), subroutines and in recursive calls, additional mechanisms are required in order to define a unique marker for occurred events.

### Checksum markers

In previous work, hardware and software instruction counters have been proposed for this purpose [10]. However, when dealing with standard commercial RTOSs we usually do not have the option to accurately save and restore the instruction counter value for each task and interrupt when they are switched in and out using the RTOS task context. Consequently a different approach is needed. For asynchronous events, a pragmatic approach of our own design, which is more generic with respect to operating systems and CPUs is to store the values of stack pointer, register-bank checksums, and/or checksums of part of the user-stack. Provided that loop-iteration counters are stored in registers, the stack-checksum is superfluous. Otherwise, a checksum of a subset of the user stack can be used, typically the part of the stack used by the currently executing function. By generating checksums we can define a unique marker: the 4-tuple

$$\langle t, PC, SP, CHKSUMS \rangle$$

where SP is the stack pointer, which differentiates between function calls. The marker is in a strict sense not unique but, as we have experienced in a great number of applications, it is a sufficient and pragmatic approximation of a unique marker. In order for the register checksums to operate properly, it is required that tasks periodically reset their registers every iteration. Concerning stack checksums, the compiler-generated code will always initialise the stack space to zero. We have successfully applied this checksum approach on a number of different platforms:

1. Processors, among them the 8/16 bit CISC Hitachi H8, the 32 bit RISC NEC V850, and the 32bit CISC Intel Pentium.
2. Compilers, among them GNU GCC (Hitachi H8, Intel x86), IAR Systems (Hitachi H8, NEC V850).
3. Real-time operating systems, among them VxWorks, and Asterix [14].

### System call markers

For synchronous events, like blocking system calls, a less elaborate approach is needed. We make use of a per-task counter, incremented each time a potentially blocking system call is invoked by that task. If the call actually blocks the task, the value of the counter is recorded as a unique marker and the counter is reset. This way we will keep track of at which system call invocation the task actually blocked.

### 5.4.2 How to Record

Recording can be performed in different ways, ranging from intrusive-free hardware and usually immobile recorders, to intrusive but mobile software recorders. In addition, there is also an option of leaving the recording mechanisms in the deployed system, with the equivalent benefit of a black-box functionality, similar to what is employed in airplanes. We describe three stereotypes of recording approaches, where the appropriateness depends on the resources available, the architecture of the target system, and whether or not black-box functionality is required.

**Type 1.** *Non-Intrusive Hardware Recorders*, use in-circuit emulators (ICE) with dual port ram. An ICE replaces the ordinary CPU; it is plugged-in into the CPU target socket, and works together with the rest of the system. The difference from an ordinary CPU is the amount of auxiliary output available. If the ICE (like those from e.g., Lauterbach, and AMC) has real-time operating system (RTOS) awareness, this type of history recorder needs no instrumentation of the target system. The only input needed is the location of the data to monitor. ICE:s have the potential to be non-intrusive since they do not steal any CPU-cycles or target memory, due to price and space limitations these cannot usually be delivered with the product. The application of this type of history recorder is consequently best suited for pre-deployment lab testing and debugging.

**Type 2.** *Software Recorders*, has an instrumented operating system and application software where the histories are stored in a number of local circular memory buffers. This type of system is intrusive in the sense that it consumes CPU cycles and memory for storage of events. One advantage of the software approach is that monitoring is performed from inside the system, wherefore on-chip memory and caches are not an issue as they might be for type 1 and type 3 recorders. It is also necessary to record data not restored during replay, e.g. externally sampled data and state variables. In contrast to the control-flow monitoring, which can be done automatically and application independent, the data-flow to monitor needs to be manually identified and tagged, using monitor wrappers. For example, `Monitor(&var, log_entry, sizeof(var_type));` During recording the monitor wrappers output the specified var to the log\_entry of the data-flow recording log. During replay the opposite occurs; var is assigned the value of the output as recorded. Since all such instrumentation will consume CPU cycles, it must remain in the target system post-deployment in order to eliminate the probe effect. This approach, in contrast to the hardware and hybrid approaches, allow for black-box functionality.

**Type 3. Hybrid Recorders.** This recorder type has hardware support and a minimum of target software instrumentation. Software probes write history data to specific addresses, and a hardware component snoops the bus at these addresses, typically in the form of a modern logic analyzer (example manufacturers are Agilent, HP, Lauterbach, VisionICE, and Microtek.) This type of recording system could also be intrusive free if all data manipulations and states were reflected in the system's external memory, and we had RTOS and data awareness (knowledge of the kernels data structures, and variable locations). However, many micro-controllers and CPUs have on-chip memory and caches, wherefore changes in state or data of the system are not necessarily reflected in the external memory. As a consequence it is necessary to perform instrumentation such that events and data monitored are recorded and stored in external memory, temporarily bypassing the cache and on-chip memory. Data-flow monitoring is similar to software recorders, with the additional penalty of a computing slowdown due to cache write-throughs and access to slower external memory. This type of history recorder is cheaper than ICE:s, but the same argumentation for not leaving the monitoring hardware in the target system still applies. There are however System on Chip (SoC) solutions [3] which can be permanently resident.

### 5.4.3 The Historian

Once the control-flow and the data-flow of the application is recorded, the first job for the historian is to sort the control-flow events in order of occurrence and to construct a timeline. A control-flow event is either asynchronous (e.g. task preemption or interrupt) or synchronous (e.g. blocking system call). For each asynchronous control-flow event, the historian generates a conditional breakpoint, such that for each PC value where asynchronous control-flow events occurred, a breakpoint is set. These breakpoints are guarded by the condition of the recorded unique marker, e.g.,

$$\langle t, PC, SP, CHKSUMS \rangle$$

For example,

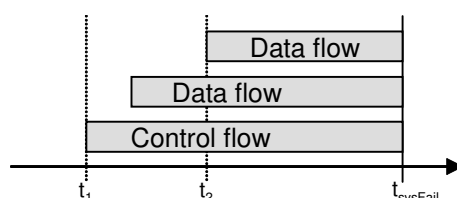
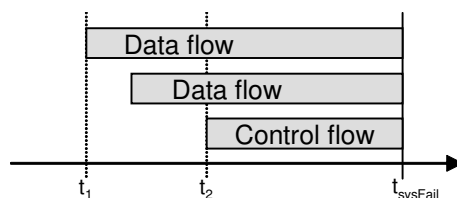
```
break at PC(event) if (SP == SP(event) &&
CHKSUMS_REGS(event) == (R0+R1+R2+Rn) &&
CHKSUMS_STACK(event) ==
(* (SP) + * (SP+1) + * (SP+2) + * (SP+m) ) )
```

Synchronous events, on the other hand, are not represented by unique individual breakpoints. Instead, the entry point of each blocking system call, used by the application, that might give rise to a synchronous event is breakpointed. The control and match of the synchronous unique marker is here managed by the Time Traveller tool, as is the transfer of control from the executing task to the subsequent task. A timeline, similar to that of the control-flow, is also assembled for the data-flow. To allow a smooth correlation between data and control-flow, both monitoring activities are closely integrated. Subsequent to the monitoring of some of the system calls, data-flow monitoring is also performed. In the following section we shall see that events mapping to these system calls are points from which the replay can be initiated. Data restoration can be handled on-target during replay, using the monitor-wrappers in the case of software- or hybrid recorders, or by the debugger environment for ICE debuggers where we can set data breakpoints, such that when a read operation is performed we can intercept it and restore the value before it is read.

#### 5.4.4 Requirements on a Starting Point for the Replay Execution

In order to make use of the data-flow- and control-flow timelines generated by the historian to achieve a deterministic re-execution, it is crucial to find a mutual starting point for the replay. In other words, we need to figure out at what control-flow log entry and at which data instance to start the replay re-execution from. The most naive approach might be to start the replay from the system start-up, where all static information of the application is known. However, this usually calls for an unreasonable long recording in order to capture the entire history from start to failure. A more reasonable approach is to make use of a set of cyclic buffers for recording, which often leaves us with the problem of having to start the replay from a non-startup state.

Since the basic idea of deterministic replay is to re-execute the application in the exact same temporal and environmental context as the recorded execution, an basic requirement is that both the control-flow- and data flow information that constitute the replay context need to be available at the start of the replay. Consider, for instance, the scenario in Figure 5.4. Due to the dimensioning of the buffers, the control-flow timeline spans from  $t_1$  to  $t_{sysFail}$ , while the shortest data flow timeline spans from  $t_2$  to  $t_{sysFail}$ . In this case, replay starting points between  $t_1$  and  $t_2$  will not be valid since no data flow information is available. Similarly, Figure 5.5 shows a scenario with an interval where no control-flow information is available.

Figure 5.4: Insufficient data-flow between  $t_1$  and  $t_2$ .Figure 5.5: Insufficient control-flow between  $t_1$  and  $t_2$ .

The requirement of available data flow at the replay starting point has to be considered when choosing how to record data. All tasks have one or more potential starting points for the replay. These starting points can be blocking system calls or delays. To start the replay of the task at a specific point for which there exists at least one entry in the control-flow log, information of the task state and inputs needs to be retrieved from the log at the re-execution of that point. Consider the task program in Figure 5.6. The task has two potentially blocking system calls (`msgQReceive` and `msgQSend`), which both, when causing task switches, are recorded in the control-flow buffer. However, only one of the calls (`msgQReceive`) has a direct subsequent data flow monitoring call, which stores (on-line) or restores (during replay) the task state and input. This fact makes this call a suitable starting point for replay, while a start from the other call has no possibility of guaranteeing a correct restoration of the task state.

As opposed to system calls, control-flow preemption or interrupt events are not suitable as replay starting points due to the fact that these events occur asynchronously and would require recording of the entire task context, in order to capture the necessary start conditions.

```
while (FOREVER)
{
    msgQReceive();
    monitor();
    ...
    msgQSend();
    ...
}
```

Figure 5.6: Suitable and non-suitable replay starting points.

### 5.4.5 The Time Traveller

By setting breakpoints at all blocking system calls, we can initialise the deterministic re-execution of the application. First, the application is reset in the debugger and the *timeline index*, an index pointing at the current control-flow event to be matched, is set to point at the first suitable starting point in the control-flow timeline. Then, each task to be replayed is executed up until it hits a breakpoint that matches a suitable starting point in the historian-generated timeline. At this point, the recorded data flow of the suspended task is written back into the application. The timeline index is incremented and the next task is set up for execution. Once the data flow of all instrumented tasks has been rewritten into the application, the replay session initialisation phase is complete.

When the initialisation is ready, the replay will step forward as the timeline index is incremented at each control-flow event successfully matched. In addition, in the event of a subsequent asynchronous event for the current task in the historian timeline, its corresponding conditional breakpoint is set, making it possible to replay this event as that breakpoint is hit. Once breakpoints representing asynchronous events are hit and successfully matched, they are removed in order to enhance the performance of the replay session. From a users point of view, this deterministic replay debug session will behave exactly like a regular sequential program mimicking the exact execution of the recorded multitasking real-time application. We can single step, insert any number of breakpoints and inspect data without introducing the probe-effect (as illustrated in Figure 5.7). We can even jump forth and back in time using the debugger (thus named the Time Machine), and define bookmarks by



generating new unique markers and new guarded breakpoints. Since we have eliminated the dependency of the external process in real-time and replaced the temporal and functional context of the application with the virtual data flow- and control-flow timelines produced by the historian, we can replay the system history repeatedly.

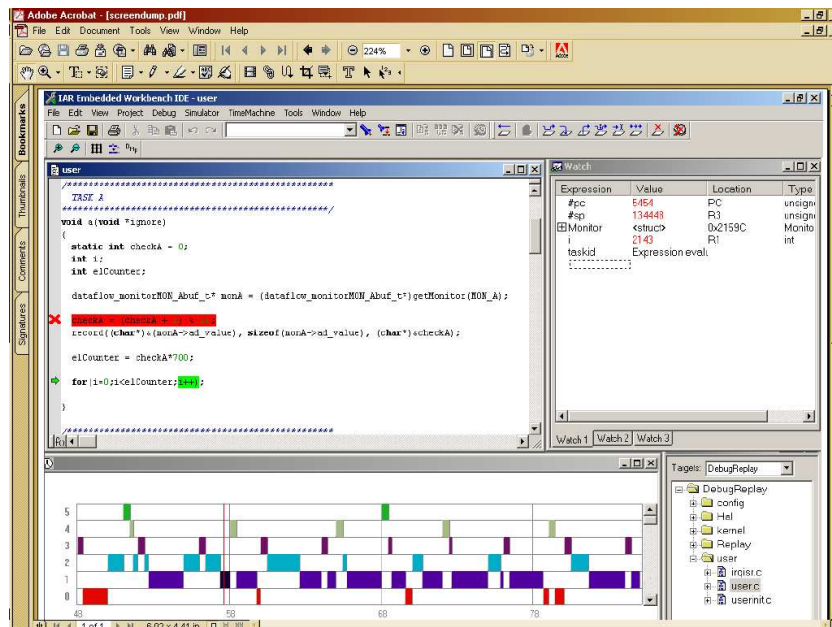


Figure 5.7: A commercial IDE with an instruction level simulator debugger, into which we have integrated our Time Machine technology (the lower left window). The time line illustrates the recorded control-flow for 6 tasks; task priorities on the vertical axis. Selecting any instance of a task re-executes the system from an idle point (the red lowest priority task) up to the selection (it is possible to jump back and forth). The debugger window shows the current state. From here it is possible to single-step, watch variables, and set new additional breakpoints.

## 5.5 Industrial Case Study

In this section, we present extracts from a case study that we have performed on an industrial robot control system. The entirety of the case study will be published in a future publication, but we provide some benchmarking results here. The developer of the investigated system is among the largest industrial robot manufacturers in the world. Their system consists of several computing control systems, signal processing systems and I/O units. We applied the Time Machine to the motion control part of the system that consists of approximately 2.5 million lines of C code and is run on the VxWorks RTOS. The subsystem for motion control is a hard real-time system, with about 70 tasks running (the most frequent task is activated every 4ms) and multiple interrupts driving an assortment of device drivers.

In our implementation a hook, which is a VxWorks feature, was programmed to capture the control-flow. Also, calls to common system calls that can change the control-flow were monitored: `msgQReceive`, `msgQSend`, `semGive`, `semTake`, and `taskDelay`. The instrumentation of these system calls was limited to a timestamp, a system call identifier, and a counter. For the system call `msgQReceive` we also included data-flow recording.

All hitherto described instrumentation was implemented totally transparent to the application code. The only manual instrumentation that had to be inserted into the application code were calls to data-flow monitors after blocking system calls, in order to capture the state of the task (represented by specified local and global variables). This amount of recording is sufficient since the code is re-executed off-line. In this case study, due to the architecture of the system, it was sufficient to capture the state after each `msgQReceive` call. Figure 5.8 illustrates the cost of monitoring in terms of execution time overhead and memory usage.

## 5.6 Future Work

We have successfully applied the time machine approach proposed in this paper in a number of applications running on different operating systems, different hardware, different compilers, and different debuggers. What we have learned however, is that it is necessary to carefully analyze the target system's dataflow with respect to what data is re-executed, re-transmitted and what data has external (process) origin in order to not forego something that may inhibit deterministic re-execution or that we do not record too much. Missing out on

<i>Monitoring activity</i>	<i>Nr of Bytes/function</i>	<i>CPU cycles (max)</i>	<i>Execution time (200MHz, Pentium II)</i>	<i><math>\Sigma</math> C/T in &amp; of CPU utilization</i>
<b>Data (monitor wrappers)</b>	360 B	1992	10 $\mu$ s	
	1024 B	3797	18 $\mu$ s	<b>0,4%</b>
	8192 B	26545	133 $\mu$ s	<b>3%</b>
<b>Task-switch</b>		378	2 $\mu$ s	<b>0,5%</b>
<b>System calls</b>	semTake, semGive, taskDelay, etc.	30	<0,2 $\mu$ s	
<b>Inter-process communication</b>	msgReceive		5 $\mu$ s	

Figure 5.8: Some benchmarking results from the industrial case study.

information is a serious problem and is something we have begun to address. We have started to look into how to automatically derive tight sets of possible data derived from what we actually have recorded. Another issue that need to be considered is replay of applications which use vast amounts of information, e.g., real-time database applications. In such applications the “state variable” (the data base) is very large and requires some kind of incremental snapshot algorithm to be manageable

## 5.7 Conclusions

In summary the contribution of this paper is a deterministic replay method for: Standard off-the shelf debuggers, and standard off-the shelf real-time operating systems. We have presented different approaches to recording, and introduced “black-box” functionality for post deployment debugging. We have presented how to make use of conditional breakpoints in standard debuggers to generate task-switches and interrupts at exactly the same location and time as recorded during runtime. We have presented a novel and pragmatic unique-marker mechanism for differentiation between loop iterations, as well as function calls. Furthermore, we have presented benchmarking results describing the overhead introduced by our solution.

# Bibliography

- [1] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of 15th Parallel and Distributed Processing Symposium*, page 10, April 2001.
- [2] P. Dodd and C. V. Ravishankar. Monitoring and Debugging Distributed Real-Time Programs. *Software – Practice and Experience*, 22(10):863 – 877, October 1992.
- [3] M. El Shobaki. A Hardware- and Software Monitor for High-Level System On Chip Verification. In *Proceedings of IEEE International Symposium on Quality Electronic Design*, pages 88 – 95. IEEE, 2001.
- [4] J. Gait. A Probe Effect in Concurrent Programs. *Software – Practice and Experience*, 16(3):225 – 233, March 1986.
- [5] S. Howard. A Background Debugging Mode Driver Package for Modular Microcontroller. Semiconductor Application Note AN1230/D, Motorola Inc., 1996.
- [6] J. Huselius. Debugging Parallel Systems: A State of the Art Report . Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.
- [7] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471 – 482, April 1987.
- [8] C.E. McDowell and D.P Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593 – 622, December 1989.

- 
- [9] M. Meerwein, C. Baumgartner, T. Wieja, and W. Glauert. Embedded Systems Verification with FPGA-Enhanced In-Circuit Emulator. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 143 – 148, 2000.
- [10] J. Mellor-Crummey and T. LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86. ACM, April 1989.
- [11] IEEE Std. IEEE Standard Test Access Port and Boundary-Scan Architecture. Technical Report 1532-2001, IEEE, 2001.
- [12] K.-C. Tai, R.H. Carver, and E.E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, 17(1):45 – 63, January 1991.
- [13] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.
- [14] H. Thane, A. Pettersson, and D. Sundmark. The Asterix Real-Time Kernel. In *Proceedings of the Industrial Session of the 13th EUROMICRO International Conference on Real-Time Systems*. IEEE Computer Society, June 2001.
- [15] J.P.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transactions on Software Engineering*, 16(8):897 – 916, August 1990.
- [16] F. Zambonelli and R. Netzer. An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications. In *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 392 – 398, 1999.



## **Chapter 6**

# **Paper C: Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies**

Daniel Sundmark, Henrik Thane, Joel Huselius, Anders Pettersson, Roger Mel-  
lander, Ingemar Reijer and Mattias Kallvi  
In Fifth International Workshop on Automated and Algorithmic Debugging  
(AADEBUG) Gent, Belgium, September 2003.

### **Abstract**

Deterministic replay is a method for allowing complex multitasking real-time systems to be debugged using standard interactive debuggers. Even though several replay techniques have been proposed for parallel, multi-tasking and real-time systems, the solutions have so far lingered on a prototype academic level, with very little results to show from actual state-of-the-practice commercial applications. This paper describes a major deterministic replay debugging case study performed on a full-scale industrial robot control system, as well as a minor replay instrumentation case study performed on a military aircraft radar system. In this article, we will show that replay debugging is feasible in complex multi-million lines of code software projects running on top of off-the-shelf real-time operating systems. Furthermore, we will discuss how replay debugging can be introduced in existing systems without impracticable analysis efforts. In addition, we will present benchmarking results from both studies, indicating that the instrumentation overhead is acceptable and affordable.



## 6.1 Introduction

ABB is a world leading manufacturer of industrial robots for industrial automation. Out of all deployed industrial robots to date, ABB has delivered about 50 percent. Out of those 50 percent, 70 percent are deployed in the car manufacturing industry. SAAB Avionics is a major supplier of electronic warfare technology on the international market. The main focus of the company is electronic warfare systems, such as display systems, tactical reconnaissance systems and electromagnetic technology services. Avionics products can for example be found in the Swedish fighter aircraft Gripen, the American F-15 and the NH- 90 helicopter.

### 6.1.1 Contribution

In this paper, we present results from two industrial case studies performed in cooperation with the above companies. With these case studies, we show that our recent research results have not merely been academic artifacts, but contributions to a fully operational method of debugging full-scale industrial real-time systems. In addition, we present benchmarking results from both case studies, showing that the overhead incorporated in the system by instrumentation is acceptable.

### 6.1.2 Paper Outline

The rest of this paper is organized as follows: Section 6.2 will give a background and a motivation to the case studies as well as short descriptions of the systems studied. Section 6.3 describes the implementations of our method in the investigated systems. In Section 6.4, instrumentation benchmarking results are presented. Finally, Section 6.5 and Section 6.6 conclude the paper and discuss future work.

## 6.2 Background and Motivation

It is no secret that testing, debugging and maintenance constitute the largest percentage of the overall cost of an average industrial software project. In a recent study, NIST [11] has shown that more than \$59 billion/year is spent on debugging software in the U.S.A. alone. As the average complexity of software applications increase constantly it is now common that testing and debugging

constitute more than 80% of the total life cycle cost [11]. A known fact is also that bugs are introduced early in the design but not detected until much later downstream in the development cycle, typically during integration, and early customer acceptance test. For embedded real-time software this fact makes the situation really difficult, since most failures that are detected during integration and early deployment are extremely difficult to reproduce. This makes debugging of embedded real-time systems costly since repetitive reproductions of the failure is necessary in order to find the bug. The lack of proper tools and methods for testing and debugging of complex real-time systems does not help the situation. The reason why ABB Robotics and SAAB Avionics systems were chosen as case study subjects was based on their high level of software- and overall technical complexity. In addition, the systems operate in safety-critical and high availability environments, where failures might be very costly, making system validation and verification even more important.

### 6.2.1 Replay Debugging

During the mid-eighties, in an effort to address the problems with inadequate tools for debugging of complex systems, LeBlanc and Mellor-Crummey proposed a method of recording information during run-time and using this information to reproduce the exact behavior of the execution off-line [9]. This method, called *Instant Replay*, allowed otherwise unfit cyclic debugging techniques to be used for debugging nondeterministic systems. Instant Replay, as many of its successors [1][4][12][13][3], was focused on debugging of non-real-time concurrent programs, thereby concentrating mainly on the correct reproduction of synchronization races and, in some cases, on-the-fly detection of data races. However, some methods for debugging of complex real-time systems have also been proposed [5][16]. These methods have called for the availability of specialized hardware and non standard instrumentation tools in order to work satisfactorily. They have also been mere academic prototypes and not suited for the complexity of real world software applications.

### 6.2.2 Real-Time System Debugging using Time Machines and Deterministic Replay

In our previous work on debugging of embedded real-time systems, we have proposed a replay method for recording important events and data during a *reference execution* and to reproduce the execution in a subsequent *replay execution* [14][15][7]. As for most replay debugging methods, if the reference

execution fails, the replay execution can be used to reproduce the failure repeatedly in a debugger environment in order to track down the bug. Our method allows replay of real-time applications running on top of standard commercial real-time operating systems (RTOS) and uses standard cyclic debuggers for sequential software. There is no need for specialized hardware or specialized compilers for the method to work and the software based instrumentation overhead has so far proven acceptable. This allows our probes to be left permanently in the system, making post-mortem debugging of a deployed system possible while at the same time eliminating the risk of experiencing *probe effects* [6] during debugging. We refer to our method as *Deterministic Replay*. The tool that is used to “travel back in time” and investigate what sequence of events that led to a failure is referred to as the *Time Machine* and consists of three major parts: The *Recorder* which is the instrumentation mechanism (similar to the black-box or flight recorder in an airplane), the *Historian* which is the post-mortem off-line analysis tool and the *Time Traveler* which is the mechanism that forces the system to behave exactly as the reference execution off-line.

The main objective of performing the case studies was to validate the applicability of the Time Machine method to existing complex real-time systems. The basic questions were:

- Would it be possible to reproduce the behavior of such complex target systems?
- How do we minimize the analysis and implementation effort required in order to capture the data required from such complex target applications?
- Would the instrumentation overhead (execution time and data bandwidth) be sufficiently low for introduction in real-world applications?

### 6.2.3 ABB Robotics System Model

To validate our ideas, we had the opportunity to work with a robotics system that is state-of-the-art in industrial manufacturing automation. The system is a highly complex control system application running on top of the commercial Wind River RTOS VxWorks. However, in order to avoid the somewhat impossible mission of analyzing and instrumenting an approximate of 2.5 million lines of code in an academic project, we focused on instrumenting five central parts in the target system:

- The operating system.

- The application's operating system abstraction layer.
- The inter-process communication abstraction layer.
- The peripheral I/O abstraction layer.
- The state preserving structures of each individual task.

All instrumentation was system wide and application-transparent except for the state preserving structures in the tasks. A more thorough description of the instrumentation will be given below. For an overview, see Figure 6.1.

Out of an approximate total of 70 tasks, we choose to record the state (internal static variables) of the three most frequently running tasks. Thus, we limited the instrumentation efforts for parts of the data flow recording. However, these three tasks constituted the major part (approximately 60%) of the CPU utilization and their data flow constituted more than 96% of the overall system data flow bandwidth.

### **Robotics System-Level Control Flow Model**

In general, task activations in the robotics system are dependent on message queues. Basically, each task is set to block on a specific message queue until a message is received in that queue. When a message is received, the task is activated, performs some action, and is finally set to wait for another message to arrive. Chains of messages and task activations, in turn, are initiated by occurrence of external events, such as hardware interrupts at arrival of peripheral input.

### **Robotics Data Flow Model**

We divide the Robotics data flow model into three parts: The per-task state preserving structures, the inter-process communication and peripheral I/O. As for the state preserving structures, each task has its own local data structure used to keep track of the current task state. This structure holds information of message data, static variables and external feedback. Naturally, this structure alters during execution as the state of the task changes.

The inter-process communication is handled by the use of an IPC layer implemented on top of the message queue primitives in VxWorks. This layer extends the functionality of the original message queue mechanisms. Finally, peripheral I/O, such as motion control feedback is received through a peripheral I/O layer.

### 6.2.4 SAAB Avionics System Model

In addition to the Robotics system, we also performed a minor case study in a military aircraft radar system. In short, the task of the system is to warn the pilot of surrounding radar stations and to offer countermeasures. This study was less extensive in that it only covered a part of the instrumentation aspect of the Time Machine technology. The full scale Saab Avionics radar system holds about 90 ADA tasks running on top of the Wind River VxWorks RTOS. However, in the scope of the case study, a reduced system with 20 central tasks was investigated.

Dataflow was limited to inter process messages. As in the robotics case, task activations in the Avionics system are controlled by message arrivals on certain queues.

## 6.3 Technique Implementations

To be able to facilitate replay, there were three things we needed to achieve: First, we needed to instrument both the VxWorks real-time kernel and the application source codes in order to be able to extract sufficient information of the reference execution. Second, we needed to incorporate the replay functionality into the WindRiver integrated development environment (IDE). We used the *Tornado 2* version of the IDE as this is the standard IDE for developing VxWorks real-time applications. Third, we needed to add the Time Machine mechanisms used to perform the actual replay of the system. Even though the Tornado 2 IDE features a VxWorks-level simulator, both recording and replay execution were performed on the actual target system. In the next sections, we will discuss these steps one by one.

### 6.3.1 VxWorks Instrumentation

A common denominator for the Robotics system and the Avionics system is that they both run on top of the Wind River VxWorks RTOS. In order to extract the exact sequence of task interleavings, it was essential to be able to instrument the mechanisms in VxWorks that directly influence the system-level control flow. Therefore, we instrumented semaphore *wait* and *signal* operations, message queue blocking *send* and *receive*, task sleep function *taskDelay* as well as preemptive scheduling decisions.

### **Blocking System Call Instrumentation Layer**

To instrument the blocking task delay-, semaphore- and message queue primitives mentioned above, we added an instrumentation layer on top of the VxWorks system call API. This layer replaces the ordinary primitives with wrappers including the added functionality of instrumentation. Apart from this instrumentation, the wrappers use the original primitives for system call operation.

### **Task Switch Hook**

With the blocking system call instrumentation layer, we are able to monitor and log what possibly blocking system calls are invoked. However, in order to see which of the invoked calls actually leads to task interleavings, we need to be able to insert probes into the scheduling mechanisms of the kernel. Fortunately, VxWorks provides several hooks, implemented as empty callback functions included in kernel mechanisms, such as a TaskSwitchHook in the task switch routine. These hooks can be used for instrumentation purposes, making it possible to monitor and log sufficient information of each task switch in order to be able to reproduce it.

### **Preemption Instrumentation**

As some of the task interleavings are asynchronous, that is, their occurrence are initiated by asynchronous events such as hardware interrupts, they do not have an origin in the logic control flow of task execution that blocking system calls do. To reproduce these events and interleavings correctly in a replay execution, we need to be able to pinpoint and monitor their exact location of occurrence. In other words, we need some sort of *unique marker* to differentiate between different program states. The program counter value of the occurrence of the event is a strong candidate for such a unique marker. However, as program counter values can be revisited in loops, subroutines and recursive calls, additional information is needed. To provide such data, we use the information of the task context available from the task control block in the TaskSwitchHook. This task context is represented in the contents of the register set and the task stack. To avoid the massive overhead introduced by sampling the entire contents of these areas, we use checksums instead [15]. Although these checksums are not truly unique, they strongly aid in differentiating between program states.

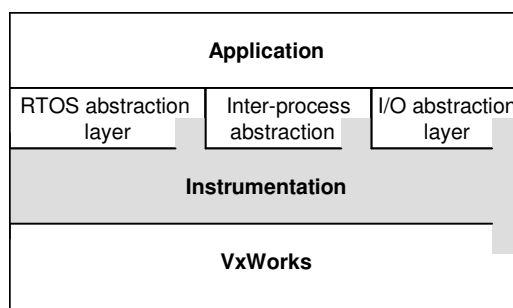


Figure 6.1: Instrumentation layer in the system model.

Other solutions to this problem have been proposed, such as hardware instruction counters [2] and software instruction counters [10], but these are not suitable in the VxWorks case due to lack of kernel instrumentation possibilities, large overheads or specialized hardware requirements.

### 6.3.2 ABB Robotics Instrumentation

In contrast to the system-level control flow instrumentation, which is handled on the VxWorks RTOS level, some of the data flow instrumentation needs to be handled on the application level. This is due to the fact that all of the data used to represent the state of the task execution need to be identified explicitly in the source code and instrumented for recording. Such data include static and global variables and structures holding information that can be altered during the entire temporal scope of system execution.

As stated in Section 6.2.3, in the Robotics system, these data are grouped together in static structures, individually designed for each task. To minimize the amount of information stored in each invocation of a task, we used filters that separated the type of data that was prone of changes during run-time from the type of data that was assigned values during system initialization and kept these values throughout the execution. The latter are not recorded during run-time. These filters were constructed from information gathered empirically during test-runs of the system.

To be able to reproduce interaction with an external context and inter-task communication, the peripheral I/O and the inter-task communication message queues are instrumented in two operating system abstraction layers, similar to that described in Section 3.1.1. This solution gives the instrumentation a

quality of transparency, making it less sensitive to changes in the application code.

However, the part of the data flow recording that is concerned with the reproduction of state preserving structures is performed by probing functions inserted at various locations in the application code. A more thorough discussion on how and where these probes should be inserted in the code is given in our previous papers [15][8]. A summarizing overview of system instrumentation can be viewed in Figure 6.1. In the figure, the gray area represents the instrumentation layer, which is slightly integrated into different parts of the RTOS and some application abstraction layers.

### 6.3.3 SAAB Avionics Instrumentation

Since both the Robotics and the Avionics system run on top of VxWorks and the RTOS-level instrumentation is application independent, the instrumentation of the Avionics system-level control flow was implemented in a very similar fashion. However, one aspect had to be taken into account. In contrast to the Robotics system, the Avionics system was implemented in Ada. As the Ada runtime environment is added as a layer on top of VxWorks, this layer had to be altered in order to be able to monitor rendezvous and other Ada synchronization mechanisms. This instrumentation allowed for the Ada runtime environment to use instrumented versions of the VxWorks synchronization system calls instead of the original versions.

As for the data flow, this study focused on inter-process communication. Messages were logged in cyclic buffers, dimensioned on a per-process basis, at the receiver. State preserving structures and peripheral I/O were not considered as in the ABB Robotics case. However, the benchmark figures of the inter-process communication recordings give a hint of the overall overhead incorporated in a full-scale instrumentation.

### 6.3.4 Time Machine

Once the code is properly instrumented, we are able to record any execution of the system in order to facilitate replay of that very execution. The next step is to implement the mechanisms of the time machine that actually performs the replay. These mechanisms were implemented in an add-on to the Tornado 2 IDE.



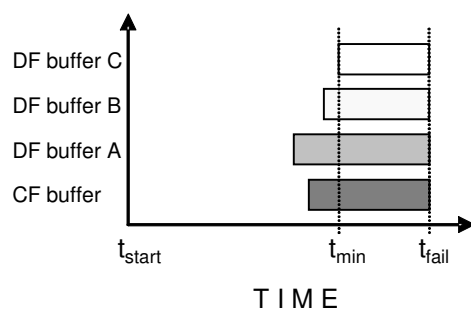


Figure 6.2: Pruning of buffer entries. Entries to the left of  $t_{min}$  are discarded.

### The Historian – Starting the Replay Execution

In our Time Machine replay system, the task of the Historian is to analyze the data flow- and system-level control flow recordings of a reference execution. We used basic cyclic FIFO buffers for recording. Combined with the fact that the cyclic buffers are of a finite length and memory resources are scarce, this rarely leads to a situation where all recorded information is available at the end of the reference execution. As these recordings most often will be of a different temporal length, some sort of pruning is needed in order to discard those entries that are out of the consistent temporal scope of all buffers. In other words, all tasks that are to be replayed needs information from both the control flow recording (one per system) and data flow recordings (one per task). Since buffers are dimensioned using a discrete number of entries and not continuous time, we will practically always end up in a situation where some buffers cover a longer span of time than others. As this information is unusable, it must be detected and discarded. This operation is performed by the Historian as depicted in Figure 6.2.

In addition, the Historian has the responsibility to find a consistent state from which the replay executions can be started [8]. Again considering Figure 6.2, if such a starting point exists, it is located in between  $t_{min}$  and  $t_{fail}$ , where sufficient information of all instrumented tasks is available. When this operation is performed, the Historian sets up the structures in the target system used to reproduce the data flow of the reference execution. A more elaborate description on how a starting state is found and a replay execution is prepared is presented in a recent paper by Huselius et. al. [8].

### The Time Traveler

As the replay execution is started, the Time Traveler interacts with the debugger and, given the information provided by the Historian and the breakpoints visited in the program, allows recreation of the system state for any given time in the scope of the replay execution [15].

### 6.3.5 IDE and Target System Integration

Tornado 2 is an integrated development environment including a text editor, a project/workspace handler, a gcc compiler, a target simulator and a gdb-based debugger, capable of performing multi-threaded debugging with no requirements on deterministic reproduction of execution orderings.

#### Tornado 2 IDE Architecture and WTX

Debugging in the Tornado 2 environment is performed by means of remote debugging. That is, the debugging session is executed on-target and controlled from the development environment via communication with an on-target task.

To handle all communication with the target system, a hostbased target server is included in the Tornado 2 IDE. All tools that seek interaction with the target system are able to connect as clients to the target server and issue requests of target operations. To provide tool vendors with a possibility to create their own add-ons to the Tornado 2 IDE, a programming interface is provided. The Wind River Tool Exchange (WTX) API enables developers to create tools that communicate and interact directly with the VxWorks target server. For the implementation of our Time Machine system, the Historian and the Time Traveler were integrated and implemented as a WTX tool, able to connect with a running target server and to force a replay execution upon the system running on that target. The structure of the Tornado 2 IDE, the Time Machine and the target system interactions is depicted in Figure 6.3.

#### Wdb Task

To handle the on-target debugging operation, VxWorks provides a dedicated task, the *Wdb task*. This task handles requests of breakpoints, watches, single-stepping and other debugging functions from the IDE. These functions are used by the Time Traveler via the WTX interface and the target server in order to control the replay execution.

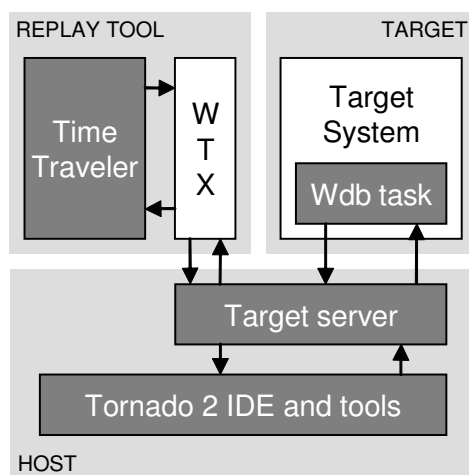


Figure 6.3: Target system, IDE and Time Machine integration.

### Breakpoints

Breakpoints play a central role in the interaction between the time machine and the target system. As described by the Deterministic Replay method [15], breakpoints are set at every point of possible task interleaving and as they are encountered in the target system, their occurrence is echoed from the Wdb task through the WTX and into the event handler of the Time Traveler. Based on the individual features of each breakpoint, the state of the replay execution can be deduced and the Time Traveler replay engine will force the desired behavior on the target system.

### Debugging Mode

Debugging in VxWorks can be performed in two different modes: Task mode and system mode. The difference is that when in system mode, an encountered breakpoint will halt the entire system, including the operating system. In task mode, a breakpoint will only halt the task that encountered it, leaving all other tasks free for execution.

Ideally, since the investigated applications are pseudoparallel, system mode debugging should be used. This would help in guaranteeing the correct ordering of events and task interleavings in the replay execution since no task is able

to continue execution and corrupt the system-level control flow if the entire system is halted. However, we experienced problems when trying to reproduce this ordering in system mode debugging regarding incapability of task suspension. This is due to the fact that no tasks can be explicitly suspended from execution by an external operation (such as requests made from the time traveler tool) in system mode debugging. In addition, the locking of the Wdb task substantially complicated communication between the target system and the IDE, making thorough investigation of the target state more difficult. Therefore, task mode debugging is used and the correct ordering of events in the replay execution is explicitly forced upon the system by means of the Time Traveler replay engine.

## 6.4 Benchmark

One of the issues of these case studies was to investigate whether the overhead incorporated by system-level control flow- and data flow instrumentation was acceptable in a full-scale complex industrial real-time application or not. In order to resolve this issue, we performed benchmarks measuring instrumentation mechanism CPU load and memory usage in both systems. Since the instrumentation is yet to be optimized and the benchmarking tests are performed under worst-case scenario conditions, many results might be rather pessimistic.

### 6.4.1 ABB Robotics

In the ABB robotics case, we timestamped the entry and the exit of all instrumentation and recording mechanisms. This gave us a possibility of extracting the execution time of the software probes. In addition, we measured the frequency and recording size of the data flow instrumentation mechanisms. The achieved results are presented in Figure 6.4. The *bytes per function* column shows the number of bytes logged by each iteration of an instrumentation function, and as each task instance has one and only one data flow instrumentation function, this figure also represents the number of bytes stored in each task instance of a task. The *CPU cycles* and the *Time* columns present the execution time spent in each instrumentation function and the *CPU utilization* column shows the percentage of all CPU time spent in each instrumentation function. Where results are left out, these are discarded due to their insignificant interference with memory or CPU utilization. We note that task 3 has a combination of a high frequency and large state preserving structures, resulting in the largest

<i>Instrumentation activity</i>	<i>Bytes per function</i>	<i>CPU Cycles</i>	<i>Time (<math>\mu</math>s)</i>	<i>CPU utilization (%)</i>
Data, task 1	360	1992	10	-
Data, task 2	1024	3797	18	<b>0.4</b>
Data, task 3	8192	26545	133	<b>3</b>
Task-switch	-	378	2	<b>0.05</b>
System calls	-	30	0.2	-
Inter-process communication	-	-	5	-

Figure 6.4: Benchmarking results from the Robotics study.

monitoring overhead (3% CPU utilization).

#### 6.4.2 SAAB Avionics

As the data instrumentation in the Avionics system is performed solely on message queues, the data flow benchmark is made in a per-queue fashion. Since there are major differences in message arrival frequencies and message size between the different queues, only the six most memory-consuming message queues are presented in the benchmark results. Out of the 17 instrumented message queues, these six consume 99% of all message queue memory bandwidth. The results of the Avionics benchmarking study are shown in Figure 6.5, presented in the form of memory utilization for logging (in the bytes/second column) and CPU utilization (in the rightmost column). As in the robotics case, it is the combination of a high frequency and a large size of data (such as the one of Queue A and Queue E) that has the most significant implications on the instrumentation memory utilization.

## 6.5 Conclusions

With this paper, we have shown that complex real-time system debugging is feasible using the *Deterministic Replay* technique and the *Time Machine* tool. This is true not only for specialized academic systems, but also for full-scale

<i>Instrumentation activity</i>	<i>Msg frequency (Hz)</i>	<i>Data size (bytes)</i>	<i>Bytes/sec.</i>	<i>CPU util. (%)</i>
Queue A	60	89	5340	<b>1.9</b>
Queue B	15	9	135	
Queue C	15	25	375	
Queue D	15	49	735	
Queue E	15	281	4215	
Queue F	15	21	315	
System-level control flow	-	-	-	<b>0.03</b>

Figure 6.5: Avionics system benchmarking results.

industrial real-time systems. Furthermore, we have shown that it is possible to achieve a high level of transparency and portability of the method by placing much of the instrumentation in system call-, inter-process communication- and peripheral I/O layers, rather than in the application source code. Both case studies presented here indicate a small CPU utilization overhead of 0.03–0.05% for system-level control flow instrumentation. The data flow instrumentation has proven more temporally substantial, but has stayed in the fully acceptable interval of 1.9–3.0%. As for memory utilization, the ABB Robotics instrumentation required a bandwidth of 2 MB/s and the Saab Avionics system called for approximately 12–15 kB/s for both system-level control flow and data flow logging. Looking at the size of these systems and the resources available, such a load is definitely affordable.

## 6.6 Future Work

We have successfully applied the time machine approach in a number of applications running on different operating systems, hardware and debuggers [15]. However, we have learned that it is necessary to carefully analyze the target system's data flow with respect to what data is re-executed, re-transmitted and what data has external origin in order not to forego something that may inhibit

deterministic re-execution or that we do not record too much. Missing out on information is a serious problem and is something we have begun to address. We have started to look into how to automatically derive tight sets of possible data derived from what we actually have recorded. Another issue that needs to be considered is replay of applications which use vast amounts of information, e.g., real-time database applications. In such applications the “state preserving variables” are very substantial and require some kind of incremental snapshot algorithm to be manageable.

# Bibliography

- [1] K. Audenaert and L. Levrouw. Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts. *Journal of Microprocessors and Microsystems, Elsevier*, 18(10):601 – 612, December 1994.
- [2] T.A. Cargill and B.N. Locanthi. Cheap Hardware Support for Software Debugging and Profiling. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82 – 83, October 1987.
- [3] J. Chassin de Kergommeaux and A. Fagot. Execution Replay of Parallel Procedural Programs. *Journal of Systems Architecture*, 46(10):835 – 849, 2000.
- [4] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of 15th Parallel and Distributed Processing Symposium*, page 10, April 2001.
- [5] P. Dodd and C. V. Ravishankar. Monitoring and Debugging Distributed Real-Time Programs. *Software – Practice and Experience*, 22(10):863 – 877, October 1992.
- [6] J. Gait. A Probe Effect in Concurrent Programs. *Software – Practice and Experience*, 16(3):225 – 233, March 1986.
- [7] J. Huselius. Debugging Parallel Systems: A State of the Art Report . Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.
- [8] J. Huselius, D. Sundmark, and H. Thane. Starting Conditions for Post-Mortem Debugging using Deterministic Replay of Real-Time Systems.



- In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS03)*, pages 177 – 184, July 2003.
- [9] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471 – 482, April 1987.
- [10] J. Mellor-Crummey and T. LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86. ACM, April 1989.
- [11] NIST Report. The economic impacts of inadequate infrastructure for software testing., May 2002.
- [12] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133 – 152, May 1999.
- [13] K.-C. Tai, R.H. Carver, and E.E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, 17(1):45 – 63, January 1991.
- [14] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.
- [15] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288 – 295). ACM, April 2003.
- [16] J.P.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transactions on Software Engineering*, 16(8):897 – 916, August 1990.



## **Chapter 7**

# **Paper D: Pinpointing Interrupts in Embedded Real-Time Systems using Context Checksums**

Daniel Sundmark and Henrik Thane  
Submitted for publication.

## **Abstract**

When trying to track down bugs in programs using cyclic debugging, the ability to correctly reproduce executions is imperative. In sequential non-real-time software, this is straightforward. However, when program execution is affected by preemptive interrupts, this will have severe effects on the ability to reproduce program behaviors deterministically, since a faithful reproduction requires exactly the same interrupt to hit the program at the exact same instruction. In previous methods, the problem of reproducing interrupts correctly has been solved using different kinds of instruction counters. However, these counters are problematic, since they induce large execution time perturbation, demand for specialized hardware or provide inexact pinpointing. This makes them highly unfit for resource-constrained embedded real-time systems. In addition, all previous methods require specialized platform-specific tools (e.g. special compilers) in order to function. In this paper, we propose an alternative method for pinpointing interrupts in embedded real-time systems using context checksums, which is not dependent on specific hardware features or special compilers - but which rather can be applied to any system.

Although context checksums in some cases also proves inexact or ambiguous, we will show that they serve as a practical method for pinpointing and reproducing interrupts in embedded real-time systems. Furthermore, our method performs perfectly well with standard development tools and operating systems, requires no additional hardware support and, in the worst case, consumes merely a tenth of the execution time of existing software-based methods for pinpointing interrupts.

## 7.1 Introduction

Cyclic debugging is the process of examining the behavior of a faulty execution in an iterative fashion, thereby narrowing down the temporal and functional scope within which the system infection (the execution of a bug) might have taken place. By doing this, the bug can hopefully be found and safely removed.

### 7.1.1 Background

Cyclic debugging is trivial as long as the programs under investigation are deterministically reproducible (i.e. their execution behavior depends only on input parameters provided and controlled by the user). However, correct reproduction of execution behavior has always been a problem during cyclic debugging of non-deterministic programs. Since such programs exhibit different execution behavior over different executions, encountered failures are hard to reproduce and therefore hard to investigate in depth.

Interrupts are a major source of non-determinism in program executions. As an interrupt occurs, the ongoing CPU activity is halted, the state of the executing program is stored and the interrupt is handled by an interrupt service routine. To reproduce this scenario (e.g. for debugging reasons), we need to be able to correctly reproduce the occurrence of the interrupt. In other words, we must make sure that the interrupt preempts the execution of the CPU activity at the exact same state during the replay as it did in the first execution.

An intuitive solution might be to make use of the program counter value at the occurrence of the interrupt. If we can make sure that the interrupt is forced upon the program at the same program counter value during debugging, the program will be interrupted at the correct instruction. However, program counter values might be revisited. For instance, a loop with ten iterations could execute each instruction in the loop during each iteration of the loop. Using solely the program counter, we have no possibility of distinguishing an interrupt occurring at some arbitrary program counter value during the third iteration of the loop from one occurring at the same instruction during the seventh iteration of the loop. Consequently, additional information is needed to uniquely pinpoint the state of occurrence of interrupts. We need some sort of a *unique marker*.

In our research, we have proposed a method, Deterministic Replay [13], for recording and reproducing (or replaying) the execution behavior of non-deterministic real-time systems. The technique presented and evaluated in this paper is incorporated into the Deterministic Replay method, allowing replay

debugging of interrupt-driven embedded real-time systems.

### 7.1.2 Related Work

Traditionally, the simplest unique marker technique is the instruction counter. In its basic form, this is a mechanism that counts machine code instructions as they are executed. When an interrupt occurs, the instruction counter is sampled along with the program counter in order to pinpoint the exact location of occurrence. The following sections will deal with the basic instruction counter as well as altered versions.

#### Instruction Counters

In order to be able to count each instruction as it is executed, there is an obvious need for some kind of specialized hardware [6][5]. Doing this in software would call for an instrumentation of the assembly code, adding an incrementing instruction after (or before) each instruction. Not only would this significantly slow down overall system performance, it would also double the size of the code part of the program (assuming all instructions are of the same size).

The hardware required to count instructions is not a complicated mechanism, but a simple counter incremented on each instruction execution cycle.

Even though the hardware instruction counter technique solves the problem of uniquely pinpointing the location of occurrence of interrupts, the method has drawbacks. One of the more significant problems is lack of the hardware needed in modern state-of-the-practice embedded microprocessors. Even though some of the larger processors available today have registers capable of performing instruction counting [1], this is no standard component in smaller or embedded processors. In addition, for many existing hardware platforms, there is reason to doubt the accuracy of the provided instruction counters [9].

Another problem is the sampling of the instruction counter. For applications using an Operating System (OS) or a Real-Time Operating System (RTOS), this may call for OS- or RTOS support. For example, consider an interrupt occurring at time  $t_{evt}$ . At time  $t_{evt} + \delta$ , the hardware instruction counter is sampled. Obviously, we will receive the instruction counter value of the latter, giving us a sampling error equal to the number of instructions executed during  $\delta$ . This might be problematic, especially if  $\delta$  varies from time to time, due to interrupt service routine- or kernel jitter.

### Software Instruction Counters

In 1989, Mellor-Crummey and LeBlanc proposed the use of a software instruction counter (SIC) [8], suitable for applications and systems running on top of platforms not equipped with the instruction counter hardware. As stated in the previous section, a software-based instruction counter performing the same task as a traditional hardware instruction counter would incur an intolerable overhead on the system. Thus, the software counterpart had to be much more restrictive when selecting upon which instructions to increment.

The SIC idea is based on the fact that only backward branches in a program can cause program counter values to be revisited. For instance, in a sequential program without backward branches, no instruction will be executed more than once. In such a system, the program counter is a unique marker, defining unique states in the execution. However, using structures such as loops, subroutines and recursive calls will require backward branches. Due to performance reasons, the implementation of the SIC not only increments on backward branches, but also on forward branches. In short, the SIC is implemented as a register-bound counter, requiring special compiler support. In addition, a platform-specific tool is used to instrument the machine code with incrementing instructions before each branch. According to the authors, the SIC incurs an execution overhead of approximately 10 % in the instrumented programs.

The problem of getting the correct instruction counter value at sampling time  $t_{\text{evt}} + \delta$  is not solved using software instruction counters, although we are only interested in the number of backward branches rather than the number of instructions during  $\delta$ .

In 2002, Kim et. al. proposed an “enhanced software instruction counter” [7], which operated similarly to the original instruction counter. The enhanced software instruction counter managed to achieve an approximate 4 – 22% overhead reduction rate (i.e., a resulting CPU utilization overhead of about 9%) by analysing the machine code and separating deterministic scopes from non-deterministic scopes, and only instrumenting the non-deterministic scopes.

In 1994, Audenaert and Levrouw proposed the use of an approximate software instruction counter [4]. Their method was denoted Interrupt Replay. In Interrupt Replay, the run-time logging of system interrupts is done by recording interrupt ID and an approximate SIC value. This version of SIC is incremented at entry- and exit points of interrupt service routines and can therefore not be used to pinpoint exact locations (program states) of occurrence of interrupts. As a consequence, Interrupt Replay is able to reproduce orderings of interrupts correctly, but the interrupts will not be reproduced at the correct instruction.

### 7.1.3 Problem Formulation

As discussed in the above section, several methods have been proposed for pinpointing and reproducing interrupts. However, due to various drawbacks, none of these methods has been fully accepted. All instruction counter methods require platform-dependent specialized development tools, such as specialized compilers, in order to function. When discussing embedded systems, additional drawbacks become significant. Very few embedded microcontrollers are equipped with sufficiently accurate hardware instruction counters. Turning to software instruction counters, these will require approximately 10 % of the overall execution time, a hard to meet requirement in resource constrained systems.

Consequently, a different approach, more adapted to the requirements of embedded systems and able to perform using standard development tools, is needed.

### 7.1.4 Contribution

In this paper, we present a novel method for pinpointing interrupts, suitable for embedded real-time systems. In our method, we use an approximation of the state of the preempted program at the time of the interrupt as a marker. This approximation is represented in the form of the stack pointer and a checksum of the execution environment of the program, such as the registers or (part of) the program stack. Our method imposes an execution time overhead of approximately 0.002 – 0.37 % (27 – 5000 times less than that of the SIC) and requires no additional hardware support to function.

### 7.1.5 Paper Outline

The remainder of this paper is organized as follows: In Section 7.2, our method is described in detail. In Section 7.3, we address the issue of the accuracy of our approximative method and in Section 7.4, we evaluate this accuracy through simulation. Here, we also evaluate the system perturbation of our method. In Section 7.5, we conclude the paper and Section 7.6 discuss future work.

## 7.2 Context Checksums

The basic idea of our method is to reproduce interrupts by recording the program counter values and unique markers of their occurrence. In order to repro-



duce these interrupts, a debugger breakpoint is set at each interrupted program counter address and the program is restarted in the debugger. As a breakpoint is hit, the unique marker of the current execution is compared to the recorded unique marker value. If these markers match, we consider this interrupt to be pinpointed and an interrupt is forced upon the system.

In this section, we will describe how our method uses the stack pointer together with approximations of the data in the execution context as unique markers. As an introduction, we will describe our concept of the execution context.

### 7.2.1 Execution Context

If we look back upon the example formulated in Section 7.1.1, where a program counter value is indistinguishably revisited a number of times, a good solution in theory would be to make use of the loop counter together with the program counter value as a unique marker. Unfortunately, not all loops have loop counters. In a more general sense, it is very hard to determine exactly which parts of the program execution context that differentiate between specific loop iterations, subroutine calls or recursive calls. Ideally, we would base our unique marker on the *entire* content of the execution context in order to be able to differentiate between loop iterations. However, considering the amount of data used to represent this context, we face a practical problem when recording it during execution due to the massive perturbation to the system. Consequently, we need to derive a subset of the execution context suitable for unique marker use.

The program execution context is basically a set of data stored in registers, on the stack or on the heap. Since the processor registers are small and very fast, these hold the most current parts of the execution context. And, since they are small and very fast, they make excellent candidates as a basis for the execution context-based unique markers.

### 7.2.2 Register Checksum

One solution would be to store the contents of each processor register. However, in most embedded systems, computing- as well as memory resources are scarce. Storing all registers at each interrupt might incur an intolerable perturbation on the memory usage of the system. In our method, we handle this problem by separately storing the stack pointer, as it is invaluable for differentiating between recursive calls, and calculating and storing a checksum of the

contents of the remaining registers. By doing this, we destroy information, but still preserve an approximative representation of the register contents from the time of the interrupt.

The register checksum operation is a simple addition of all processor registers. Overflow of the accumulated checksum is ignored. Hence, if the processor is equipped with eight general-purpose 16-bit registers ( $R_0 \dots R_7$ ), the register checksum  $C_R$  is calculated as follows:

$$C_R = (R_0 + R_1 + \dots + R_7) \text{ mod } 2^{16}$$

Due to the modest size and the ease of access of processor registers, the computational cost of calculating a register checksum is very small. However, since the register checksum is based solely on the processor registers, its main disadvantage is that it only covers a minor subset of the execution context. If an interrupt occurs within a loop and the actual parameters differentiating between iterations are not included in this subset, we will not be able to uniquely pinpoint the occurrence of the interrupt.

### 7.2.3 Stack Checksum

In order to capture those interrupt occurrences not successfully pinpointed by the register checksum, we must expand the interval of execution context included in the context checksum.

As we already used the registers, the remainder of the execution context is located on the stack and on the heap. In our method, we chose to work with the program stack contents. This has two reasons: First, implementing a stack checksum calculation in an instrumentation probe [10] is significantly easier than implementing a heap checksum in the same probe. The stack area is well defined, continuous and often easy to access from within the probe. Second, without having extensive proof of this, we assume that variables influencing the program control flow, such as loop counters, are often allocated on the stack rather than on the heap.

The checksum operation of the stack checksum is identical to the one performed in order to calculate the register checksum. Here, the subset of the execution context included in the checksum is bounded by the boundaries of the stack of the executing program. Hence, on a 16-bit architecture, the stack checksum  $C_S$  is calculated using the following formula:

$$C_S = (S_{SP} + S_{SP+1} + \dots + S_{SB}) \text{ mod } 2^{16}$$

In the above formula,  $S_X$  denotes the byte at stack address  $X$ .  $SP$  denotes the value of the stack pointer at the time of the interrupt and  $SB$  denotes the value of the stack base of the interrupted program.

The stack checksum should be viewed upon as a complement to the register checksum rather than a stand-alone solution for pinpointing interrupts. The reason for this is the fact that the execution overhead of the stack checksum exceeds the overhead of the register checksum to such an extent that the perturbation of the latter becomes negligible. In addition, discarding the option of using a register checksum when choosing a stack checksum solution will eliminate the possibility of detecting changes in register-bound variables over loop iterations.

### Instrumentation Jitter

Apart from the sheer size issue, there is another property that separates the register checksum from the stack checksum. In the case of the register checksum, we always use the same number of elements in order to calculate the checksum. If the processor has eight registers, the checksum will always calculate the register checksum by accumulating the values stored in these eight registers. Hence, we can guarantee a constant execution time as far as number of instructions are concerned.

Using a stack checksum, the situation is different. The stack base of a program is constant whereas the stack pointer varies over time. This implies a variable size of the program stack and thus a variable execution time of the stack checksum calculation, depending on the size of the stack at the time of the interrupt.

Variations in execution time of software are usually referred to as *jitter*. In multi-tasking systems (such as most embedded real-time systems), designers try to keep the jitter to a minimum, since it comprises the testability and analyzability of the system [11]. Therefore, jitter introduced by instrumentation activities (such as the stack checksum calculation) may complicate testing of sensitive systems, even though the instrumentation was included in order to increase the analyzability.

#### 7.2.4 Partial Stack Checksum

To reduce the execution time perturbation and the problem of large instrumentation jitter when using the stack checksum technique, the developer has the option of not including the entire program stack in the stack checksum. A

partial stack checksum  $C_P$  would be calculated similarly to the original stack checksum (once again on a 16-bit platform):

$$C_P = (S_{SP} + S_{SP+1} + \dots + S_{SX}) \text{ mod } 2^{16}$$

However, the upper boundary  $SX$  of the stack interval to be included in the checksum is chosen such that:

$$SP \leq SX \leq SB$$

By using this formula, we once again reduce the percentage of the execution context included in the stack checksum, thereby reducing the accuracy of the unique marker approximation. In turn, we obtain the following benefits:

- **Eliminating instrumentation jitter**

By defining  $SX$  in terms of a constant positive offset to  $SP$  (denoted  $x$  in Figure 7.1) such that the interval  $[SP, SX]$  delimits a constant number of bytes on the stack, we make sure that the stack checksum will be calculated using a constant number of instructions. This will eliminate the instrumentation jitter of the checksum calculation (not considering cache effects or similar). In the case where the size of the fixed interval  $[SP, SX]$  exceeds the size of the actual program stack (i.e. when  $SX > SB$ ), this can be detected and the remaining instructions can be simulated using additions of zero to the checksum or similar.

- **Reducing instrumentation overhead**

Intuitively, reducing the percentage of the stack included in the stack checksum will reduce the execution time of the stack checksum calculation. If total elimination of instrumentation jitter is no major requirement, a reasonable candidate for  $SX$  may be the base of the stack for the current subroutine. Many processors are equipped with a dedicated register holding the value of this base pointer ( $BP$  in Figure 7.1) to the current scope of execution.

### 7.3 Approximation Accuracy

In the above section, we have proposed a set of unique marker approximations designed to be able to pinpoint locations of occurrence of interrupts. As our method is inexact, this raises questions of how accurate these approximations

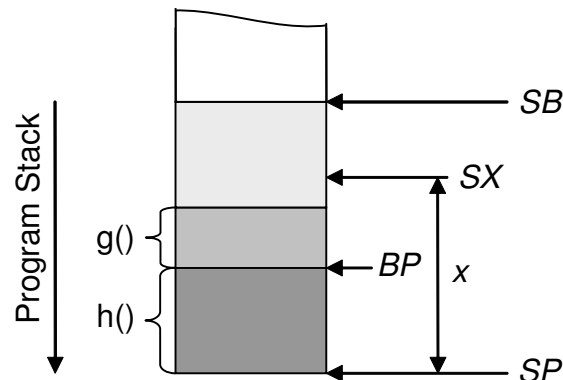


Figure 7.1: Different delimiter alternatives for the stack checksum. The  $g()$  and  $h()$  intervals represent stack intervals for subroutine execution scopes for  $g$  and  $h$  respectively.

are. In this section, we will discuss the ambiguity of our method, starting by describing our method in a more formal notion.

Given a program  $P$ , we define  $S_P$  to be the set of states that can be reached in an arbitrary execution of  $P$ . Each element  $s \in S_P$  is made up of a tuple  $\langle pc_s, env_s \rangle$ , where  $pc_s$  is an executable address in the program code and  $env_s$  is a reachable program environment. Furthermore, an execution  $E_P$  of  $P$  is defined as an ordered set of visited states. Hence, in a more formal notion, given an execution  $E_P$  that is preempted by an interrupt at state  $s_i$ , our aim is to uniquely identify  $s_i$ .

If no information is extracted during the execution, we know nothing of when the interrupt preempted the program. As far as we know, the interrupt may have occurred anywhere during the execution. In other words, there is no element  $s \in E_P$  that can be disqualified from being the potential state of the interrupt.

By identifying the program counter of the state of the interrupt  $pc_i$ , we are able to eliminate a large subset of the set of visited states in the program execution. Those states for which  $pc \neq pc_i$  can be discarded from further investigation. However, we are still left with a set of inseparable states, since we cannot differentiate between the various state environments.

Adding the stack pointer and the register- and stack checksum will aid in further pruning the execution state set. Using these, we have access to an approximative representation of the program environment of  $s_i$ .

Using these prunings, we will end up with a non-empty set  $E_i$  for which the following is true:

$$E_i = \{s : s \in E_P \wedge pc_i = pc_s \wedge env_i \approx env_s\}$$

Ideally, at this stage  $E_i$  will include exactly one element (the actual state of the interrupt). Unfortunately, we cannot guarantee that the environment approximation and the program counter value are not valid for other states in  $E_P$ . The reason for the inability of differentiating between  $s_i$  and other states in  $E_P$  is twofold:

- **Insufficient scope of execution context**

Intuitively, including a smaller scope of execution context in the context checksum will increase the risk of leaving important variables out. Thus, a register checksum alone will provide less accuracy than a register checksum combined with a stack checksum. If we are dependent on variables located on the heap, which are addressed by memory direct machine code operations, typically possible in a CISC architecture, neither register- nor stack checksums will be of any use.

- **Checksum ambiguity**

As the checksum by default is a non-reversible operation, two completely different stacks or sets of registers may give rise to the exact same checksum. For example, both  $1 + 3 + 5$  and  $7 + 2 + 0$  equals 9, even though they contain entirely different terms. In addition, since overflow is handled in no other way than a simple modulo operation, our method will not differ between a checksum of 1 and one of  $2^n + 1$ , where  $n$  is the number of bits in the checksum.

It should be noted that our method will not fail in pinpointing the correct state of the interrupt  $s_i$ . The problem is that it also might find *false positives*, i.e. it might pinpoint other states as well. Due to the fact that the interrupt-matching set  $E_i$  is ordered (states are ordered in the same sequence as in which they were visited in the original execution), in our current implementation, the method will choose the state that is reached first. Yet the question remains, how frequently do we find the correct state?

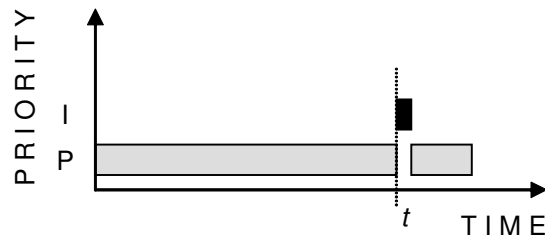


Figure 7.2: Execution of program  $P$  preempted at time  $t$  by interrupt  $I$

## 7.4 Simulation

In order to evaluate the approximation accuracy and the level of perturbation of our method, we have conducted a number of tests. In the accuracy tests, a tailor-made program  $P$  was written, executed and preempted by an interrupt  $I$ . Our test platform was the IAR Embedded Workbench (EW) [2], a commercial integrated development environment for embedded systems. We used the NEC V850 (a RISC architecture processor) version of EW and our tests were performed using the Deterministic Replay implementation on the EW V850 target simulator. Using the cycle counter of the simulator, we were able to simulate interrupts after a fixed number of clock cycles.

In our experiments, both  $P$  and  $I$  were implemented as real-time tasks running on top of the Asterix real-time operating system [12]. By varying the time  $t$  (or rather the number of clock cycles during  $t$ ), we can cause  $I$  to preempt  $P$  at different states of its execution (see Figure 7.2). The accuracy tests and their results are described further in Section 7.4.1. As for the perturbation tests, these were performed on a full-scale industrial robotics application [10] running on top of an Intel PII 400 MHz processor and the commercial VxWorks real-time operating system [3]. These tests will be further discussed in Section 7.4.2.

There are several reasons for choosing different test platforms for different experiments. In a way, it would be desirable to use the full-scale industrial application for the accuracy tests as well. However, using a tailor-made program instead, it is possible to force execution scenarios upon the method in such a way that it is tested more thoroughly. In addition, due to the current implementations, applications running on top of Asterix are significantly more manageable with respect to interactive debugging. This is an invaluable prop-

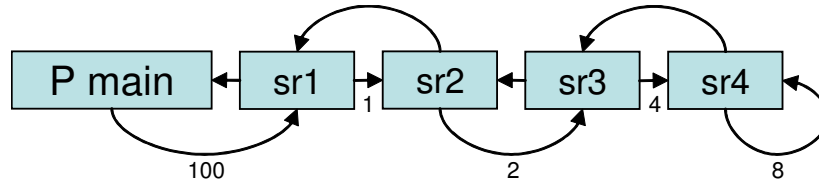


Figure 7.3: The structure of test program  $P$ .

erty when examining and comparing program states during run-time.

The perturbation tests were performed on an actual 2.5 million LOC industrial application with approximately 70 tasks in order to produce results relevant to other existing industrial applications.

#### 7.4.1 Approximation Accuracy

To be able to test the accuracy of the context checksum methods under different circumstances, the program  $P$  was written such that its properties could be easily changed. Over all tests, however,  $P$  preserved its basic structure (depicted in Figure 7.3). In short,  $P$  consists of a main function and four subroutines  $sr1..sr4$ . In a loop with 100 iterations,  $P$  main calls  $sr1$ , which in turn performs some calculations and calls  $sr2$ . The  $sr2$  subroutine, in turn, calls  $sr3$  in two loop iterations. From  $sr3$ ,  $sr4$  is called four times. In its structure,  $sr4$  has a loop that iterates eight times.

On the lowest subroutine level, this yields 6400 iterations ( $100 * 1 * 2 * 4 * 8$ ) in the  $sr4$  loop for each execution of  $P$ , meaning that each instruction in this loop is visited 6400 times. As a consequence, in an execution of  $P$ , every instruction of the  $sr4$  loop will result in 6400 states in  $E_P$ . All of these states will have identical program counter values, but different environments. Hence,  $P$  is well suited for examination of how well our method will perform regarding differentiation between states with identical program counter values. In our tests, we used four different allocation schemes in order to investigate the performance of our method. These schemes were modeled such that the allocation of variables were placed in different parts of the execution context:

1. Stack Allocation 1

In this scheme, all variables (loop counters and calculation variables) were allocated on the stack of each subroutine. No parameters were passed.



2. Stack Allocation 2

This scheme allocated like Stack Allocation 1, but also passed parameters explicitly between subroutines.

3. Heap Allocation 1

In this scheme, all variables were allocated globally. No parameters were passed.

4. Heap Allocation 2

All variables were allocated globally and parameters were passed between subroutines.

Each of these schemes were tested using the register checksum, the stack checksum and the partial stack checksum. In our test, we used the scope of the current subroutine as a delimiter of the partial stack interval (corresponding to the  $[SP, BP]$  interval in Figure 7.1). All in all, this yielded 12 different test scenarios. These scenarios were tested by executing  $P$  during  $t$  time units. At time  $t$ ,  $P$  is preempted by an interrupt and the unique marker and the program counter are sampled. Then,  $P$  is deterministically re-executed until the program counter and the unique marker matches those that were sampled. At this point, we compare the current state of execution with the original interrupt state by comparing the values of a set of globally defined loop counters. If the states match, we consider the test successful.

As stated in Section 7.4, by varying  $t$ , we can cause the interrupt to preempt the program at different states each time. In our tests, we started at a  $t$  value of 10000 clock cycles and in increments of 200, we raised it to 18000 cycles. This produced 41 test cases in each of the 12 test scenarios. The reason for the sparse number of tests is that they had to be performed by hand.

Each test case yields a binary outcome (either the interrupt is successfully pinpointed, or it is not). Looking at the cumulative success rate percentage in relation to the number of test cases, most test scenarios exhibit a benign curve (see Figure 7.4). However, some curves do not seem to converge nicely after 41 test cases (e.g. Figure 7.5). For these test scenarios, we doubled the number of test cases in order to get a more stable success rate. For graphs from all simulations, see Appendix A.

In Figure 7.6, the results of the accuracy simulations are shown. Naturally, since the partial stack checksum and the stack checksum are complementary techniques to the register checksum, these always exhibit a better accuracy. If all variables are allocated on the stack, the stack checksum techniques will

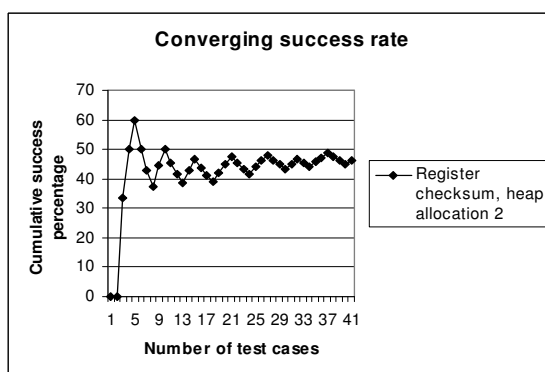


Figure 7.4: For this test scenario, the cumulative success percentage converges after 41 test cases.

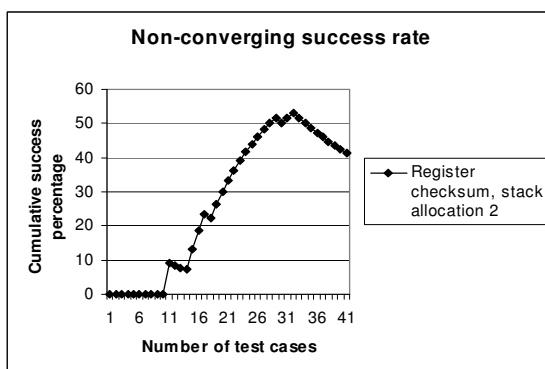


Figure 7.5: This test scenario gives rise to a non-converging cumulative success rate. After 41 test cases, it is hard to tell where the curve is going. Consequently, more test cases are needed.

	Stack allocation 1	Stack allocation 2	Heap allocation 1	Heap allocation 2
Register checksum	42%	45%*	54%	46%
Stack checksum	100%	100%	68%*	73%
Partial stack checksum	100%	93%	59%*	49%

Figure 7.6: Success rates for different unique marker techniques. Those figures marked with an asterisk are based on 82 test cases rather than 41.

outperform the register checksum techniques by far. However, if all variables are allocated on the heap, the difference is not that significant. It should also be noted that a checksum of parts of the stack in many cases performs nearly as well as a full stack checksum.

### 7.4.2 Perturbation

As the unique marker checksums need to be sampled during run-time at the occurrence of an interrupt, this imposes a perturbation to the execution of the context switch. However, contrary to the perturbation of the SIC, discussed in Section 7.1.2, the size of this perturbation is not proportional to the number of branches in the code, but to the number of interrupts in an execution.

In order to test the level of perturbation of our method, we measured the execution-time perturbation of the checksum calculations in a full-scale industrial robotics system [10]. While letting the system perform some simple robot arm movements, we sampled the unique markers at interrupt occurrences as well as the execution time of the unique marker code. The upper and lower boundaries of the instrumentation execution time is presented in Figure 7.7.

As we can see from the figure, the instrumentation jitter of the stack checksum execution is several magnitudes larger than that of the register checksum. The alterations in stack checksum execution time are mostly due to differences in the size of the stack at the time of the interrupt, whereas the register checksum execution time alterations are due to cache effects (since the registers are sampled from the task control blocks rather than the actual hardware registers).

Regarding the level of perturbation, this could be compared with that of

	Register checksum	Stack checksum
WCET	1.75 $\mu$ s	0.11 ms
BCET	0.37 $\mu$ s	0.40 $\mu$ s
Overall perturbation	0.002 - 0,007%	0.003 - 0.37%

Figure 7.7: Perturbation levels for stack- and register checksum.

the software instruction counter [8], which requires approximately 10 % of the overall CPU utilization.

## 7.5 Conclusions

In this paper, we have presented an alternative approach for pinpointing interrupts in embedded real-time systems. The need for non-standard tools for previous methods leads to problems when trying to port these methods to different platforms, processors, operating systems or compilers. Furthermore, these methods suffer from drawbacks such as insufficient hardware support [6][5], inexact pinpointing of interrupts [9][4] or large execution time perturbation [8].

For our method, simulations suggest a worst-case success rate performance of 42 % for the register checksum, 49 % for the partial stack checksum and 68 % for the stack checksum. As more variables are allocated on the stack, the performance increases significantly. In the best test cases, both the stack checksum and the partial stack checksum produce perfect results. This is achieved with an instrumentation perturbation at least ten times lower than that of the software instruction counter. Furthermore, our method only makes use of standard compilers, operating systems and platforms and requires no specialized hardware in order to function correctly.

---

## 7.6 Future Work

Considering a full program execution, we might have to deal with pinpointing not only one, but maybe two, four, or even hundreds of preemptive interrupts. Suppose that an execution is preempted by  $n_i$  interrupts. If the probability of pinpointing one interrupt correctly is  $P_i$  and the interrupts are uncorrelated, the probability of reproducing the entire execution with all interrupts in place is  $P_i^{n_i}$ . With large numbers of interrupts, the probability of correctly reproducing the entire execution will be unacceptably low. Therefore, we will look further into possibilities of significantly raising the probability of pinpointing sequences of interrupts.

## Appendix A

In this section, we present graphs from all test scenarios. All graphs display the evaluation of the cumulative success rate with respect to the number of test cases run. Figures 7.8 through 7.10 display the cumulative success rate curve of the original register-, stack- and partial stack checksum tests respectively. Figure 7.11 displays the evaluation of the cumulative success rate for the three test scenarios that were simulated with the double number of test cases.

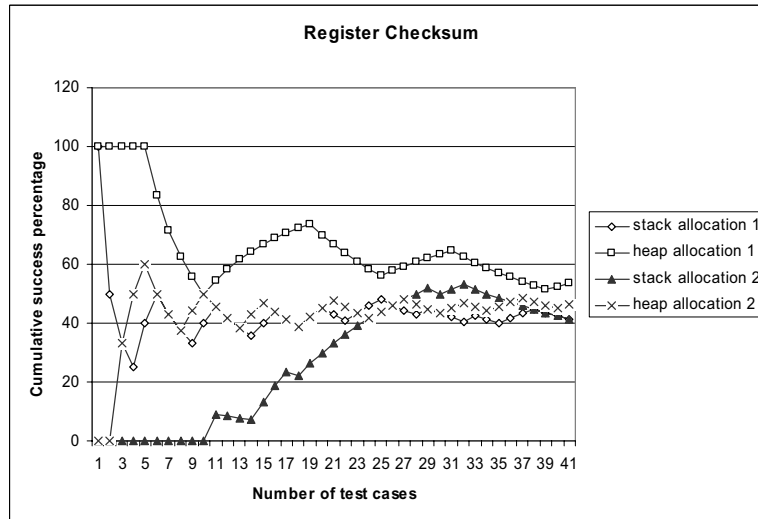


Figure 7.8: Cumulative success rate of the register checksum simulations.

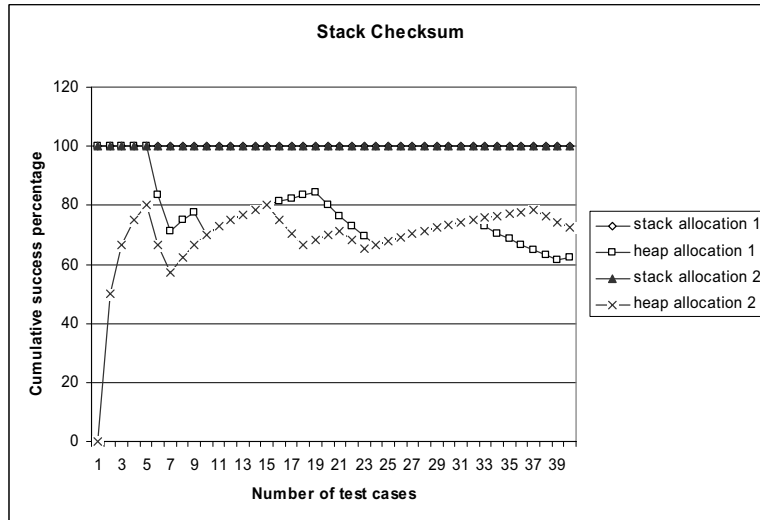


Figure 7.9: Cumulative success rate of the stack checksum simulations.

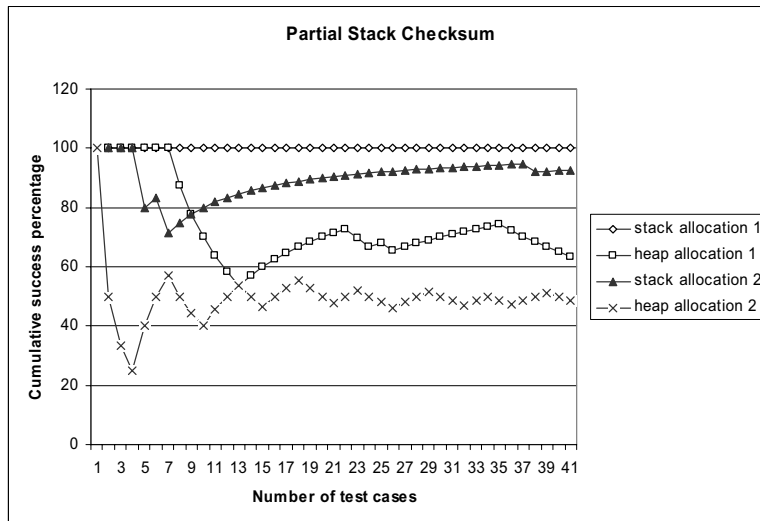


Figure 7.10: Cumulative success rate of the partial stack checksum simulations.

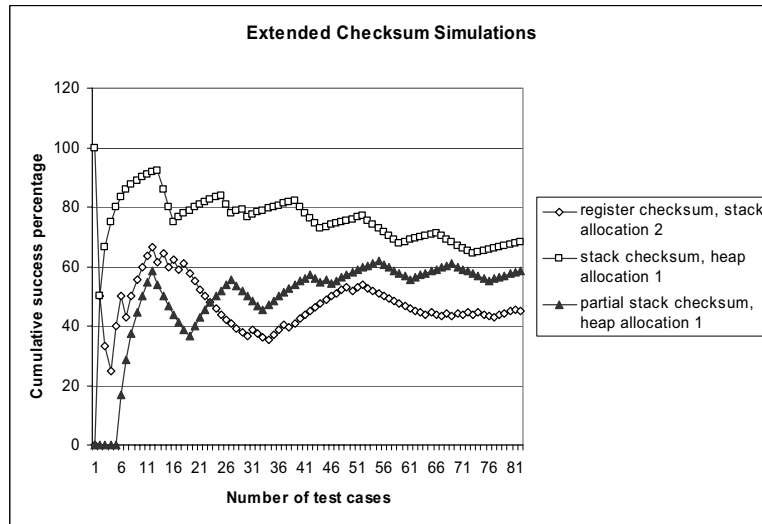


Figure 7.11: Cumulative success rate of the extended checksum simulations.



# Bibliography

- [1] *Intel Architecture Software Developer's Manual 24319202*, 1999.
- [2] [www.iar.com](http://www.iar.com), February 2004.
- [3] [www.windriver.com](http://www.windriver.com), February 2004.
- [4] K. Audenaert and L. Levrouw. Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts. *Journal of Microprocessors and Microsystems, Elsevier*, 18(10):601 – 612, December 1994.
- [5] T.A. Cargill and B.N. Locanthi. Cheap Hardware Support for Software Debugging and Profiling. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82 – 83, October 1987.
- [6] M. Johnson. Some Requirements for Architectural Support of Debugging. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 140 – 148. ACM, March 1982.
- [7] D. Kim, Y.-H. Lee, D. Liu, and A. Lee. Enhanced Software Instruction Counter Method for Test Coverage Analysis of Real-Time Software. In *Proceedings of IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.
- [8] J. Mellor-Crummey and T. LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86. ACM, April 1989.

- 
- [9] O. Oppitz. A Particular Bug Trap: Execution Replay Using Virtual Machines. In *M. Ronsse, K. De Bosschere (eds), proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG)*, pages 269 – 272. Computer Research Repository (<http://www.acm.org/corr/>), September 2003.
- [10] D. Sundmark, H. Thane, J. Huselius, A. Pettersson, R. Mellander, I. Reiyer, and M. Kallvi. Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies. In *M. Ronsse, K. De Bosschere (eds), proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG)*, pages 211 – 222. Computer Research Repository (<http://www.acm.org/corr/>), September 2003.
- [11] H. Thane and H. Hansson. Testing Distributed Real-Time Systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24:463 – 478, February 2001.
- [12] H. Thane, A. Pettersson, and D. Sundmark. The Asterix Real-Time Kernel. In *Proceedings of the Industrial Session of the 13th EUROMICRO International Conference on Real-Time Systems*. IEEE Computer Society, June 2001.
- [13] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288 – 295. ACM, April 2003.



