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

Daniel Sundmark  
dsk@mdh.se

October 2002

Abstract

Testing and debugging are major parts of a software development project, counted in time, 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 where nearly all software, desktop or embedded, grows increasingly more complex. Unfortunately, it has 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.

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).

Now, we have come accross 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 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.

Figure 1: Infection, incubation period and fault propagation

Although there today 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.

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.

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.
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.

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].
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?

2.1 Determinism and Reproducability

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 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.

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

Figure 2: A sequential deterministic program

Thane [29] gives a more formal description of determinism and reproducability 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. Reproducability, 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 reproducability, 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 with cyclic debugging of non-deterministic systems.

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 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 3: A basic interaction with a known and controllable external context](image)

This might seem somewhat trivial, but as we continue our line of reasoning and consider the system shown in Figure 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.

2.2.1 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, because of the fact that reproducibility of a deterministic system calls for a total control of the parameters that defines 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 infinite. Therefore, environment simulators will not be further discussed in this report.

### 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 of truly parallel systems introduces additional complexity compared to debugging of pseudoparallel systems, the main focus of this report will be on the latter. However, in the remainder of this paper 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 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 an 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 5.

![Figure 5: A correct ordering of events in system $S$](image)

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 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.

![Figure 6: A faulty ordering of events in system $S$](image)
2.3.1 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 inter-leavings, preemptions and interrupts in a multi-tasking system. The system-level control flow could also include invocations of semaphore primitives or other synchronization mechanisms. If we were to look back upon Figure 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.

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
\texttt{semaphore\_release} operation). In contrast, asynchronous interrupts can occur anywhere
outside sections protected by \texttt{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 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 synchroniza-
tion of these clocks, they will differ with a precision of, say, \(2\delta\). Given this, ordering an event \(e_1\)
occurring at local time \(t\) in processor \(A\) and an event \(e_2\) occurring at local time \(t + \delta\) 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:

\[
\text{Timing} \implies \text{Ordering, whereas Ordering} \not\implies \text{Timing}
\]

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 3.4), but all proposals aim at achieving \textit{race detection} and not \textit{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 and
the shared time base of the system and the external context is lost. Problems of this nature,
when the intrusively examining of a system affects the system behavior itself, are denoted \textit{Probe
Effects} [10], and will be discussed later in Section 4.1.
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 7, these simulators run as an application program on the debugger host.

![Figure 7: Using simulator-based debugging, the target simulator runs on the host](image)

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 8, the JTAG [25] and BDM [11] standards are examples of the latter.

![Figure 8: Using JTAG and BDM standards for remote debugging](image)

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.
Figure 8: Remote debugging uses the actual target hardware for debugging

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 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 reproducability to a negligible level. As depicted in Figure 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.

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 reproducability 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 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 context (or rather a peripheral device) can actively interact with the system by means of an asynchronous interrupt. This interrupt most
Figure 10: Execution of a multi-tasking real-time system, dependent on the temporal- and the external context within which it is executing.

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 reproducability, 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.
3 Debugging by Execution Replay

As the reproducability 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. The previous 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 execution 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.

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 was denoted Instant Replay and focused 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 could then 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 has to be serialized such that for any two writes \( W_1 \) and \( W_2 \):

\[
W_1 \rightarrow W_2 \lor W_2 \rightarrow W_1
\]

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 program \( P(X) \) 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 considers 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 3.3 and 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 6.

Considering the list of requirements presented in Section 2.6, the above proposals lack significant 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 in a 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 one single purpose, it is often desirable to perform a system-level debugging rather than a program-level debugging.

### 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 are at least ten years old. In 1989, Banda and Volz 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, 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.

### 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 11. Two tasks, A and B share a mutual resource and this resource is accessed within critical sections, protected by semaphores in the code for
Figure 11: Faulty execution due to erroneous event ordering

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 $B$.

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 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 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 $t_4$. Unfortunately, the outcome of this ordering of events does not violate the system specification.
3.3.1 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 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.

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

Figure 13: Interrupt occurs while in a loop

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 so that it uniquely defines the state of the program at the occurrence of the event.

3.3.2 Unique Marker Techniques

Previous work aimed at pinpointing the location of occurrence of asynchronous events has focused 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 differentiate 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 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 counter 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 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.

3.4 On-The-Fly Race Detection

In Section 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 is denoted RecPlay and 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 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.

4.1 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 15: An execution leading to a system failure](image)

The cause of probe effects are best described by an example. Consider the two-task system in Figure 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 \textit{first} access by $B$, \textit{then} access by $A$. As we can see from Figure 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 \texttt{printf}-statement), represented by the black section in Figure 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 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.

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 itself in the form of probe effects.

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 a well-defined behavior. 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 increase the ability to debug the system has 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.

Figure 16: The same execution, now with an inserted software probe, “invalidating” the failure
5 Deterministic Replay

In Section 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.

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 of 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.

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.
6 Replaying Long-Running Applications

When we listed our requirements on a real-time replay method way back in Section 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.

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 such an execution would be infeasible for several reasons. For instance, consider the long-running application in Figure 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.

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 only focuses on reproduction of message communication in a concurrent message-passing system.

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, well, 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.

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 DeBoosschere 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.

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, both 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.
References


