Replay Debugging of Real-Time Systems Using Time Machines

Henrik Thane, Daniel Sundmark, Joel Huselius and Anders Pettersson
Mälardalen University
Mälardalen Real-Time Research Centre,
Västerås, Sweden
{henrik.thane}@mdh.se

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 e.g., loop iterations 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 strength 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.

Keywords
Real-time systems, distributed real-time systems, determinism, debugging, replay, monitoring, probe-effect, black box.

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.

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 sensors values from the reference execution (the first run)?

Second, how do we keep the shared time base intact? During the debugging 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 example, an ABS-breaking system in a car. During the testing phase, a failure is discovered. To reproduce the failure and to find
the bug causing the failure the system is re-executed in a debugger, while at the same time 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 examining the state of the system.

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. Furthermore, as can be seen in Figure 1, the insertion and removal of diagnostic probes can cause non-deterministic races. This is known as the probe effect [5][11], were the actual act of observation changes the behavior of the subject of study.

1.3 Debugging by the use of Time Machines

We will in this paper present a debugging technique based on deterministic replay [3][10][18][19], which we call the Time Machine. In contrast to previous work in the field (including our own [19]) our method does not require special debuggers, modified operating systems or code transformations to work. Standard off-the-shelf debuggers and real-time operating systems suffice. 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. We will show how the system can be debugged both forward and backwards in time using standard debuggers like CPU instruction level simulators, JTAG[8] debuggers, BDM[7] debuggers or In Circuit Emulators (ICE) debuggers, with a timing precision corresponding to the exact machine code instructions at which the events occurred.

1.4 Contribution

The contributions of this paper can in essence be summed up to:

- 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 a recent industrial case study were we successfully applied our Time Machine Technology to a 2.5 million lines of code Industrial Robot controller.

Paper outline: Section 2 provides an overview of related work Section 3 presents the system model and section 4 the method for deterministic real-time systems debugging. Benchmarking results are presented in Section 5. Finally, in Sections 6 and 7 we conclude and give some hints on future work.

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 [20][3], or on special compilers generating dedicated instrumented code [13][3], which have 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 (after code transformations), but not real-time specific events like preemptions, asynchronous interrupts or mutual exclusion operations [18][1][24]. For a more elaborate discussion on related work see [21].
An early version of our deterministic replay technique, which supported replay of interrupts, preemption of tasks and distributed transactions, has been presented previously [19]. 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. It also relied on a unique marker not suitable for standard components. In this paper we generalize the previous result to standard components.

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-tasking systems with real-time kernels that support preemptive on-line scheduling.

We define control-flow events in the system to be either synchronous or asynchronous. Synchronous events are defined by potentially blocking system calls of which we know the entry and exit, e.g., semTake(). Asynchronous events are defined by scheduled preemptions or interrupt hits, for which it is not possible to a priori know where they may change the system control-flow.

We further assume that we have access to either instruction level simulator debuggers, JTAG debuggers [8], BDM debuggers [7] or In Circuit Emulator (ICE) debuggers [22]. 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).

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.

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.

4.1.1 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 transpired 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 2), subroutines and recursive calls, additional information is required in order to define a unique marker for occurred events.

Checksum markers

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

Figure 2. The PC is not sufficient as a unique marker.
```
In previous work, hardware and software instruction counters have been proposed for this purpose [13]. 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, since we do not have access to the source code. 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 different makes of CPUs is to store the values of stack pointer, register-bank checksums, and/or parts of the checksums 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, or task. By calculating a checksums at asynchronous event, we can define a unique marker: the 4-tuple <t, PC, SP, CHKSUMS>, 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 sufficient and pragmatic approximation of a unique marker. In order for the register checksums to operate properly, it is required that tasks reset their registers for every activation period (defined by e.g., a while(overflow) loop). 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 Hitatchi H8, the 32 bit RISC NEC V850, and the 32bit CISC Intel Pentium.
2) Compilers, among them GNU GCC (Hitatchi H8, Intel x86), IAR Systems (Hitatchi H8, NEC V850).
3) Real-time operating systems, among them WindRiver’s VxWorks, and Asterix [23].

System call markers

For synchronous events 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.

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 (typically implemented as a cyclic buffer). We describe three types 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 interfaces with the rest of the system. The difference from and 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 (knowledge of task control block structure and location), 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 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 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 [4] which can be permanently resident.

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., \(<t, PC, SP, CHKSUMS>\).

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.

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 naïve 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, a 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 3a. Due to the dimensioning of the cyclic 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 0-7695-1926-1/03/$17.00 (C) 2003 IEEE
t will not be valid since no data flow information is available. Similarly, Figure 3b shows a scenario with an interval where no control-flow information is available.

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 task activations. 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 4. 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.

### 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 user’s 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). 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

```c
While(FOREVER)
{
   msgQReceive();
   monitor();
   ...
   msgQSend();
   ...
}
```

Figure 4. Suitable and non-suitable replay starting points

![Figure 5. 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.](image)
application with the virtual data flow- and control-flow timelines produced by the historian, we can replay the system history repeatedly.

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, ABB Robotics, 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 could 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 offline. In this case study, due to the architecture of the system, it was sufficient to capture the state after each msgQReceive call. Figure 6, illustrates the cost of monitoring in terms of execution time overhead and memory usage. In summary the total cost in terms of control-flow recording was about 0,05% of processor utilization, for dataflow recording it was less than 4% with about 2 MB/s in terms of data usage.

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 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.

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 also validated its feasibility in an industrial case study. 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

<table>
<thead>
<tr>
<th>Monitoring activity</th>
<th>Nr of Bytes/func.</th>
<th>CPU cycles (max)</th>
<th>Execution time (200MHz, Pentium II)</th>
<th>Σ CPU in % of CPU utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data (monitor wrappers)</td>
<td>360 bytes</td>
<td>1992</td>
<td>10µs</td>
<td></td>
</tr>
<tr>
<td>1024 B</td>
<td>3797</td>
<td>18µs</td>
<td>0,4%</td>
<td></td>
</tr>
<tr>
<td>8192 B</td>
<td>26545</td>
<td>133µs</td>
<td>3%</td>
<td></td>
</tr>
<tr>
<td>Task-switch</td>
<td>378</td>
<td>2µs</td>
<td>0,05%</td>
<td></td>
</tr>
<tr>
<td>System- calls</td>
<td>semTake, taskDelay, etc.</td>
<td>30</td>
<td>&lt;0.2µs</td>
<td>0,005%</td>
</tr>
<tr>
<td>Inter process Communication</td>
<td>msgQReceive</td>
<td>5µs</td>
<td>0,1%</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. Some benchmarking results from the industrial case study.
results from the industrial case study.

8 REFERENCES


