Debugging Using Time Machines
Replay Your Embedded Systems History

Henrik Thane and Daniel Sundmark

Mälardalen Real-Time Research Centre,
Department of Computer Engineering
Mälardalen University, Västerås, Sweden,
henrik.thane@mdh.se

Zealcore Embedded Solutions AB,
Hemdalsvägen 17, 723 35 Västerås, Sweden,
henrik.thane@zealcore.com

Abstract

Cyclic debugging is one of the most important and most commonly used activities in program development. During cyclic debugging, the program is repeatedly re-executed to track down errors when a failure has been observed. This process necessitates reproducible program executions. Applying classical debugging techniques such using breakpoints or single stepping in real-time systems change the temporal behavior and make reproduction of the observed failure during debugging less likely, if not impossible. Consequently, these techniques are not directly applicable for cyclic debugging of real-time systems.

In this paper we present how do you turn standard CPU instruction level simulator debuggers, JTAG/BDM debuggers or in circuit emulator debuggers into veritable time machines so you can debug your embedded application both forwards and backwards in time – repeatedly. By on-line recording significant system events, and off-line deterministically replaying them, we can inspect the real-time system in great detail while still preserving its real-time behavior.

Keywords: Determinism, debugging, monitoring, probe-effect, testing, distributed real-time systems, replay, black-box, instruction simulators.

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, e.g., by infecting variables, memory, etc, and finally the infected state propagates to outputs. 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. This requires knowledge of the start conditions and a deterministic execution. 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 distributed real-time programs. We will briefly discuss the main issues by making the transition in three steps:

Debugging sequential real-time programs

Moving from single-tasking non-real-time programs to single-tasking real-time programs adds the concept of interaction with, and dependency of, an external context. The system can be equipped with sensors, sampling the external context and actuators, interacting with the context. In addition, the system is equipped with a real-time clock, giving the external and the internal process a shared time base. If we try to debug such a program, we will encounter two major problems: First, how do we reproduce the readings of sensors done in the first run? These readings need to be reproduced in order not to violate the requirement of having exactly the same inputs to the system. Second, how do we keep the shared time base intact? During the debug phase, the developer needs to be able to set breakpoints and single-step through the execution. However, breaking the execution will only break the progress of the internal execution while the

external process will continue. Consider, for instance, an ABS-breaking system in a car. During the testing phase, a failure is discovered and the system is run in a debugger. While the system is run in the debugger, the testing crew tries to reproduce the erroneous state by maneuvering the vehicle in the same way as in the first run. However, breaking the execution of the system by setting a breakpoint somewhere in the code will only cause the program to halt. The vehicle, naturally, will not freeze in the middle of the maneuver and the shared time base of the internal and external system is lost. This makes it impossible to reproduce the failure deterministically and simultaneously thoroughly examining the state of the system at different times in the execution.

A mechanism, which during debugging faithfully and deterministically reproduces these interactions, is required.

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 instead of one, the tasks will interact with each other both in a temporal and a functional manner. Kernel invocations and hardware interrupts will change the flow of control in the system. In addition, tasks sharing resources leads to the problems with critical regions and race conditions. Consider a system with two tasks, *A* and *B*, both sharing the resource *X*. In a test run shown in Figure 1, *A* beats *B* in a race situation for *X* and this leads to a failure.

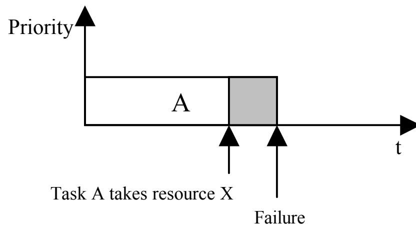

Figure 1. Task *A* takes resource *X* before being preempted by *B*.

The developer tries to investigate what led to the failure and inserts some kind of software probe in the system in order to monitor what happened. When this probe executes, it extends the execution-time such that *B* beats *A* in the same race that *A* won in the first execution. This scenario is illustrated in Figure 2.

This time, the execution does not encounter any failure and the cause of the first failure is still unknown. This type of behavior, when the insertion and removal of probes affect the execution of the system, is called the probe effect [3].

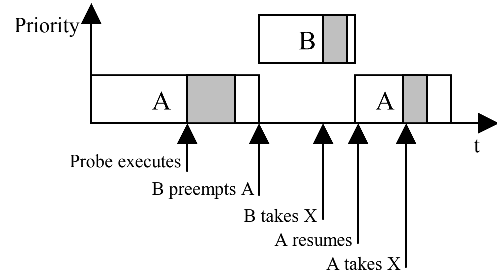

Figure 2. Task *B* preempts *A* before *A* takes resource *X*.

In moving from debugging sequential real-time programs to debugging multitasking real-time programs executing on a single processor we must consequently have mechanisms for reproducing task interleavings, and races between the executing tasks.

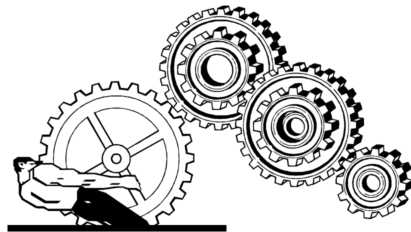

Figure 3 Classic break-pointing and single-stepping techniques violates the timing behavior of the inputs, outputs and the executing tasks in a real-time system.

Debugging of distributed real-time systems

The transition from debugging single node real-time systems to debugging distributed real-time programs introduces the additional problems of correlating observations on different nodes and break-pointing tasks on different nodes at exactly the same time.

To implement distributed breakpointing we either need to send *stop* or *continue* messages from one node to a set of other nodes with the problem of nonzero communication latencies, or we need *á priori* agreed upon times when the executions should be halted or resumed. The latter is complicated by the lack of perfectly synchronized clocks, meaning that we cannot ensure that tasks halt or resume their execution at exactly the same time. Consequently, a different approach is needed.

Debugging by the use of Time Machines

We will in this paper present a debugging technique based on deterministic replay [17][1][8][14][17], which we call the TimeMACHINE. During runtime, information is recorded with respect to interrupts, task-switches, timing, and data. The system behavior can then be deterministically reproduced off-line using the recorded history, and inspected to a level of detail, which until now has been unprecedented. The TimeMACHINE uses standard debuggers and CPU instruction level simulators

or JTAG or BDM debuggers as well as In Circuit Emulators (ICE) without the risk of introducing temporal side effects. Interrupts, task-switches and data can be reproduced and the system debugged both forward and backwards in time, with a timing precision corresponding to the exact machine code instructions at which the events occurred.

Deterministic replay is useful for tracking down errors that have caused a detected failure, but is not appropriate for speculative explorations of program behaviors, since only recorded executions can be replayed.

We have adopted deterministic replay to single tasking, multi-tasking, and distributed real-time systems. By recording all synchronization, scheduling and communication events, including interactions with the external process, we can off-line examine the actual real-time behavior without having to run the system in real-time, and without using intrusive observations, potentially leading to probe-effects [3]. We can thus deterministically replay the task executions, the task switches, interrupt interference and the system behavior repeatedly. This also scales to distributed real-time systems with globally synchronized time bases, or to systems with deterministic broadcast buses like e.g. CAN [2]. If we record all interactions between the nodes we can locally on each node deterministically reproduce them using JTAG or BDM debuggers, ICE debuggers or instruction level simulator debuggers and then globally correlate them with corresponding events recorded on other nodes.

Contribution

The contribution of this paper is a method for debugging real-time systems, which to our knowledge is

- *The first method for deterministic debugging of single tasking and multi-tasking real-time systems using standard debuggers.*
- *A refinement to the first method for deterministic debugging of distributed real-time systems by Thane and Hansson[17].*

Paper outline: *Section 2* presents our system model and *Section 3* our method for real-time systems debugging. *Section 4* provides a small example to illustrate the method. *Section 5* discusses some general issues related to deterministic replay. *Section 6* discuss different approaches to monitoring and recording of information from the target system. *Section 7* gives an overview of related work. Finally, in *Section 8*, we conclude and give some hints on future work.

2 The System Model

We assume a distributed system consisting of a set of nodes. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units for sampling and actuation of an external process. We further assume the existence of a global synchronized time base [2][4] with a known precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

The software that runs on the distributed system consists of 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, message passing and shared memory.

We assume for each node an execution strategy ranging from an interrupt driven single program system to a run-time system with real-time kernels that supports preemptive scheduling.

We further assume that we have either instruction level simulator debuggers, JTAG/BDM debuggers or ICE debuggers available We assume that the debuggers have scripting languages or equivalent interfaces such that macros or programs can be invoked conditionally at specified breakpoints.

3 The Time Traveling process

There are three basic elements to this revolutionary debugging technology

1. **The Recorder,** is a mechanism that collects all the necessary information regarding task-switches, interrupts, and data.
2. **The Historian,** is the system that that automatically analyzes, and correlates events and data in the recording, and compose these into a chronological timeline of breakpoints and predicates.
3. **The Actor,** deterministically replays the history in the debugger by generating interrupts, task-switches and restoring data as defined by the timeline.

This process is performed without ever changing the target executable code. The same code (including real-time operating system) that is run in the target, during runtime, is run during replay in the instruction level simulator.

We will now in further detail discuss and describe our method for achieving time travel and deterministic replay.

We follow the structure in the introduction and start by giving our solution to handling sequential software with real-time constraints, and then continue with multitasking real-time systems, and distributed multitasking real-time systems. We will then continue with a discussion on different approaches to recording and how you extract information from your embedded system.

3.1 Debugging single task real-time systems

Debugging of sequential software with real-time constraints requires that the debugging is performed such that the temporal requirements imposed by the environment are still fulfilled. This means, as pointed out in the introduction, that classical debugging with breakpoints and single-stepping cannot be directly applied, since it would invalidate timely reproduction of inputs and outputs – *You cannot breakpoint the world.*

However, if we identify significant variables, like state variables, and peripheral inputs like readings of A/D converters or events like accesses to the local clock, and record them we can off-line replay them. We only need to run a historian off-line that constructs a timeline of the recorded data and events. Using ordinary debuggers we, off-line automatically, "short-circuit" all identified variables, inputs or events according to the recorded timeline, i.e., we substitute readings of actual values with the recorded values.

This enables us to eliminate the time dependency of the system and replay the systems history over and over. We can even jump forth and back in time using the debugger (thus the name the TimeMACHINE), while still allowing the insertion of an arbitrary number of breakpoints and watches without introducing the probe-effect.

3.2 Debugging multitasking real-time systems

To replay and debug multitasking real-time systems we need, in addition to the data flow that is recorded for single task real-time systems, to record the system control flow. Essentially this corresponds to the task switches and the interrupt interference, i.e., the transfers of control from one task to another task, or from one task to an interrupt service routine and back. To identify these events we record where and when they occurred using timestamps and the program counter (PC). However, since PC values can be revisited in loops (Figure 4), and subroutine as well as recursive calls, and due to the coarse and unpredictable granularity of regular timers in CPUs it is necessary to define a *more* unique marker.

If the target processor supports instruction counters (IC) the unique marker can be defined by the tupel <t, PC, IC>. However, since instruction counters are not very common in commercial embedded micro-controllers, we

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

Figure 4. The PC is not sufficient as a unique marker.

need another approach. An alternative approach is to make use of software instruction counters (SIC) [10] that count backward branches and subroutine calls in the assembly code. However, these counters requires the compiler manufacturer to provide this feature (which they do not) or special target specific tools that scan through the assembly code and instrument all backward branches and subroutine calls. The approach also affects the performance, since it usually dedicates one or more CPU registers to the instruction counter, and therefore reduces the possibility of compiler optimizations.

In our approach we make use of a much simpler and much more efficient software based method that can be applied to any processor and operating system without the need for special processor features, special compilers or tools. However due to patenting issues we cannot disclose how…

The off-line historian does in addition to what it does for the data flow in single tasking real-time systems create a timeline of all task-switches and interrupt hits. This timeline is an ordered list of breakpoints for each recorded event. The historian also generates a program for each breakpoint which resets the system to the state it had during runtime. For example, one such breakpoint program resets the real-time kernel scheduler such that upon a simulated/generated timer interrupt the scheduler starts the *recorded* task, or it generates an interrupt as recorded.

3.3 Debugging distributed real-time systems

To deal with distributed systems or multiprocessor systems we simply put separate recorders on each node. For each local recording we run a historian that derives a timeline for each node.

As we by design can record significant events like I/O sampling and inter-process communication, we can on each node record the contents and arrival time of messages from other nodes. The recording of the messages therefore makes it possible to locally replay, one node at a time, the exchange with other nodes in the system without having to replay the entire system concurrently. Globally synchronized time stamps of all events make it possible to

debug the entire distributed real-time system, and enables visualizations of all recorded and re-executed events in the system.

Alternatively, to reduce the amount of information recorded we can off-line re-execute the communication between the nodes. However, this requires that we order-wise synchronize all communication between the nodes, meaning that a fast node waits up until the slow node(s) catch up. This can be done truly concurrently using several nodes using JTAG/BMD or ICE debuggers, or simulated on a single host computer using an instruction level simulator for each node.

Global states

In order to correlate observations in the system we need to know their orderings, i.e., determine which observations are concurrent, and which precede and succeed a particular event. In single node systems or tightly coupled multiprocessor systems with a common clock this is not a problem, but for distributed systems where there is no common clock this is a significant problem. An ordering on each node can be established using the local clocks, but how can observations between nodes be correlated?

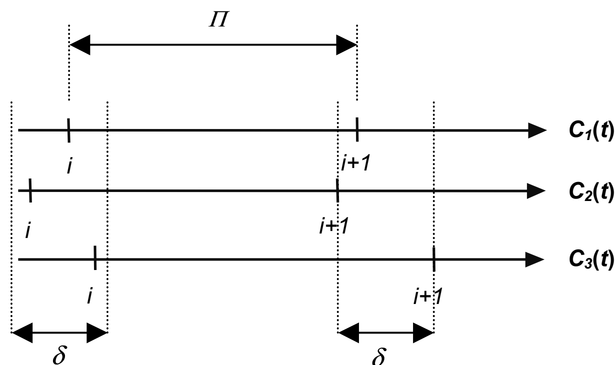

Figure 5 The occurrence of local ticks on three nodes

One approach is to establish a causal ordering between observed events, using for example logical clocks [7] derived from the messages passed between the nodes. However, this is not a viable solution if tasks on different nodes work on a common external process, without exchanging messages, or when the duration between observed events is of significance. In such cases we need to establish a total ordering of the observed events in the system. This can be achieved by forming a synchronized global time base [2][4]. That is, we keep all local clocks synchronized to a specified precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

Figure 5 illustrates the local ticks in a distributed system with three nodes, all with tick rate Π, and synchronized to the precision δ. There is no point in having $\Pi \leq \delta$, because the precision δ dictates the margin of error of clock readings, and thus a $\Pi \leq \delta$ would result in overlaps of the δ intervals during which the synchronized local ticks may occur [6].

Consider Figure 6, illustrating two external events that all three nodes can observe, and which they all timestamp. Due to the sparse time base [5] and the precision δ, we end up with timestamps of the same event that differ by 1 time unit (i.e., Π) while still complying with the precision of the global time base. This means that some nodes will consider events to be This concurrent (i.e., having identical time stamps), while other nodes will assign distinct time stamps to the same events. is illustrated in Figure 6, where node 2 will give the events *e1* and *e2* identical time stamps, while they will have difference 2 and 1 on nodes 1 and 3, respectively. That is, only events separated by more than 2Π can be globally ordered.

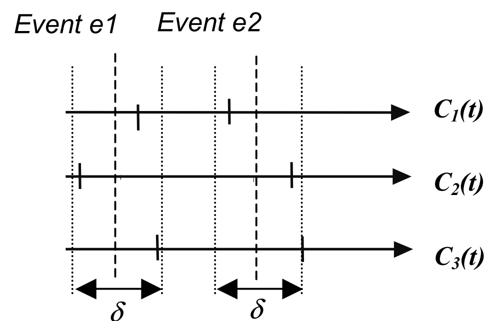

Figure 6. The effects of a sparse time base.

4 A small example

We are now going to give an example of how the entire recording and replay procedure can be performed. The considered system has four tasks *A*, *B*, *C*, and *D* (Figure 7). The tasks *A*, *B*, and *C* are functionally related and

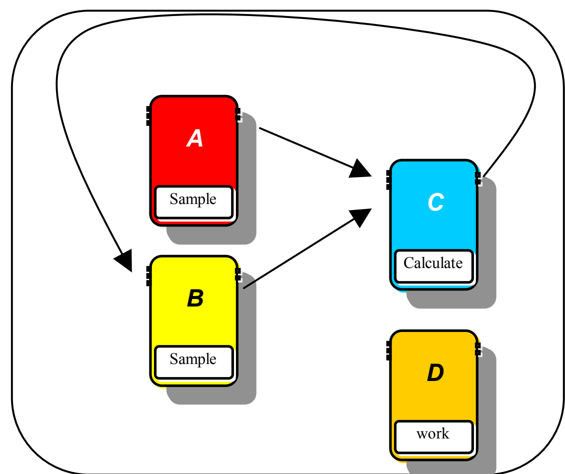

Figure 7. The data-flow between the tasks

exchange information. Task *A* samples an external process via an analog to digital converter (A/D), task *B* performs some calculation based on previous messages from task *C* and samples an external process, and task *C* receives both the processed A/D value and a message from *B*; subsequently *C* sends a new message to *B*.

Task *D* has no functional relation to the other tasks, but preempts *B* at certain rare occasions, e.g., when *B* is subject to interrupt interference, as depicted in Figure 8. However, task *D* and *B* both uses a function that by a programming mistake is made non re-entrant. This function causes a failure in *B*, which subsequently sends an erroneous message to *C*, which in turn actuates an erroneous command to an external process, which fails. The interrupt *Q* hits *B*, and postpones *B*'s completion time. *Q* causes in this case *B* to be preempted by *D* and therefore becomes infected by the erroneous non-reentrant function. This rare scenario causes the failure. Now, assume that we have detected this failure and want to track down the error.

We have the following control transfer recording for time 50 -117:

1. Task *A* starts at time 50
2. Task *A* stops at time 55
3. Task *B* starts at time 60
4. Interrupt *Q* starts at time 70, and preempts task *B* at PC=*x*
5. Interrupt *Q* stops at time 72
6. Task *B* resumes at time 72, at PC=*x*
7. Task *D* starts at time 80, and preempts task *B* at PC=*y*
8. Task *D* stops at time 87
9. Task *B* resumes at time 87, at PC=*y*
10. Task *B* stops at time 89
11. Task *C* starts at time 100
12. Task *C* stops at time 117

Together with the following data recording:

1. Task *A* at time 51, read_ad() = 234
2. Task *B* at time 60, message from *C* = 78

The historian then generates conditional breakpoints at location *x*, and *y* as well as programs that cause the Interrupt *Q* to occur at *x* and the preemption of task *B* by task *D* at location *y*. Task *A*'s access to the read_ad() function is short circuited and fed with the recorded value instead. Task *B* gets at its start a message from *D*, which is recorded before time 50.

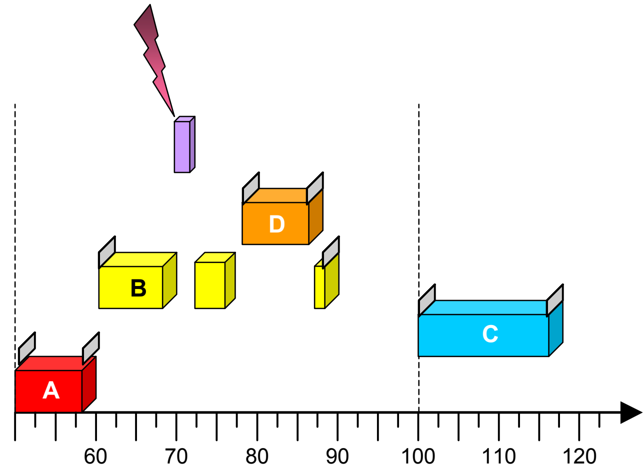

Figure 8. The recorded execution order scenario

The message transfers from *A* and *B* to *C* is performed by the kernel in the same way as it would on-line.

The programmer/analyst can breakpoint, single step and inspect the control and data flow of the tasks as he or she see fit in pursuit of finding the error. Since the replay mechanism reproduces all significant events pertaining to the real-time behavior of the system the debugging will not cause any probe-effects.

As can be gathered from the example it is fairly straightforward to replay a recorded execution. The error can be tracked down because we can reproduce the exact interleavings of the tasks and interrupts repeatedly. Experience has shown that reproducing failures of the exemplified kind is very difficult in practice. A deterministic replay mechanism is thus an invaluable tool.

5 Recording

With respect to recording we have several options ranging from intrusive-free hardware and immobile recorders, to intrusive but deterministic software and mobile recorders. We also have the option of leaving the recording mechanism in the deployed system, with the equivalent benefit of a black-box (as in airplanes). If the deployed system crashes we can extract the information from the black-box and use it for deterministic replay of the system up to the crash. We are going to describe three

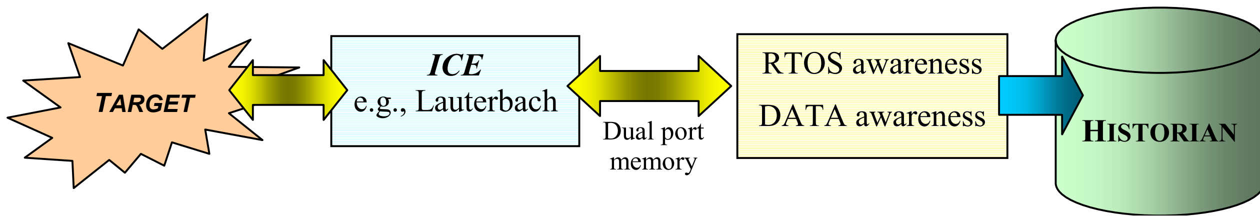

Figure 9. Monitoring via dualport memories and in circuit emulators.

stereotypes. However, which one is most suitable depends on the application.

Type 1. ***Non Intrusive Hardware Recorders***. Hardware in-circuit emulators using dual port ram are used (e.g., Lauterbach, AMC, etc.)

This type of history recorder need no instrumentation whatsoever of the target system, if the in circuit emulator (ICE) has real-time operating system (RTOS) awareness, or interrupt service routine awareness. Many commercial ICEs (like e.g., Lauterbach, AMC, etc) provide this functionality. That is, no instrumentation is needed and you can observe the state of the RTOS by monitoring the changes in the data structures of the RTOS via the dual port memory. The same can be done for simpler event triggered systems by observing interrupt occurrences, and data (I/O, communication and access to memory) can be recorded if you know the location of it in memory, of which most compilers and linkers can provide.

This type of history recorder is non-intrusive since it will not steal any CPU-cycles or target memory. One drawback however is that this type of system cannot be delivered with the deployed target system (the black-box) since its too expensive, especially for high volume systems like car subsystems or consumer electronics. Another con is that these systems are hard to expand to multiprocessor and distributed systems due to cost and synchronization issues. The application of this type of history recorder is consequently best suited for pre-deployment lab testing and debugging.

Type 2. ***Hybrid Hardware Software Recorders.*** This recorder type has hardware support and a minimum of target software instrumentation. Hardware in-circuit emulators or logic analyzers collect histories using bus snooping and instrumented software. (e.g., Agilent, Lauterbach, VisionICE, Microtek, etc.)

This type of recording system could also be intrusive free if all data manipulations and states were reflected in the systems external memory, and we had RTOS and data awareness. However, many micro-controllers and CPUs have on-chip memory and caches which means that any changes in state or data of the system is not reflected in the external memory. In the latter case it is necessary to instrument the operating system such that interrupts and task-switches are recorded and stored in external memory, bypassing the cache. Many existing RTOSs have "hooks" that can be used for instrumentation. The same goes for data and internal state of the application. The cost for this overhead is roughly 10-20 read write operations, and about 12-20 bytes of data stored at every task switch, depending on the CPU. The external historian then collects all changes in this buffer and constructs a history.

This type of history recorder is cheaper than an ICE, but has the same problem of scalability with respect to multiprocessor systems and distributed systems as the type 1 recorder above. Likewise it is not possible to leave the monitoring hardware monitoring mechanisms in the target system (the black-box) due to cost and size issues. Specially designed co-processors could however be deployed with the target system to some expense. The

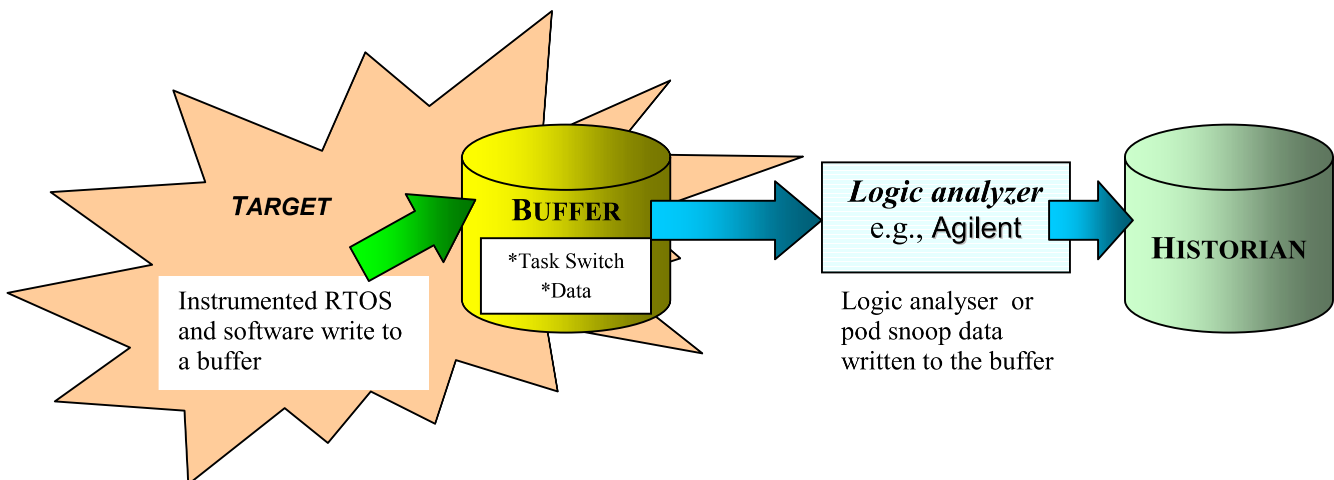

Figure 10. Monitoring via bus snooping and instrumented software.

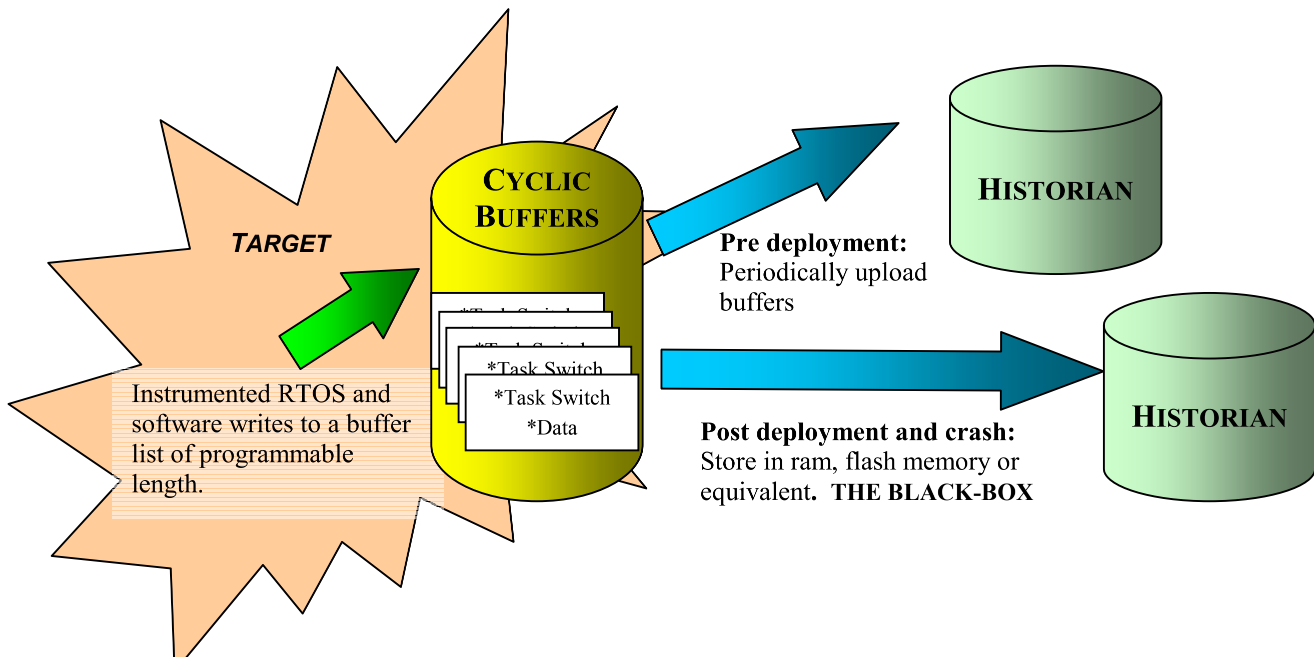

Figure 11. Monitoring via instrumented RTOS and application software.

application of this type of history recorder is consequently also best suited for pre-deployment lab testing and debugging.

Type 3. ***Software Recorders***. The target operating system and software is automatically instrumented for storage of histories in circular memory buffers of programmable length.

This type of system is intrusive in the sense that is consumes CPU cycles and memory for storing task switch information and application data, but it is deterministic since the intrusion is constant, i.e., what is run during testing and debugging is run in the delivered system. The instrumentation consists of a cyclic series of buffers in the target RTOS and software for storage of tasks-switches, interrupts, timing and application data, as well as code for storing these events and data. In essence it is a *type 2* system with more memory.

During testing you periodically upload the contents of the cyclic buffers (history) to the historian for assembly of longer histories, or you run the system until if crashes and then upload the history stored in the target to the historian.

This approach, in contrast to the hardware and hybrid approaches, makes it possible for us to diagnose and debug the system after deployment (the blak-box). This means that when a customer reports a failure, you can relive the recorded history and diagnose what happened. You can travel back in time and deterministically reproduce exactly what happened during runtime. Examples of suitable application areas are safety-critical systems, like those in the automotive industry and medical systems. More examples are robotic production lines, or other systems were production stop costs plenty of money and where it is not possible to stop the entire system for debugging.

The cost for this software recording approach is an overhead of roughly 10-20 read/write operations, and about 12-20 bytes of data for every task switch depending on the CPU. Each buffer entry contains data of an event, for example, who started, who preempted, who terminated, who resumed, etc? At which program counter value did it happen, at what time, etc? The memory cost is a function of the length of the recording and the size of the buffer entries. Typically this amounts to ca 0.2kB – 2kB (10 – 100 events). It is also necessary to record the data that cannot be restored off-line by re execution, e.g., sampling via A/D converters, state, messages received via a communication network, access to the real-time clock, etc. Note, that it is not necessary to store data, like messages passed between tasks since these transmission can be re executed off-line. The memory need for data storage is also a function of the length of the recording, but also dependent of the size of the data.

Software recording is also a great deal cheaper for multiprocessor and distributed systems applications, than the hardware of hybrid approaches.

6 Discussion

Statement: One can only replay what has previously been observed, and no guarantees that every significant system behavior will be observed accurately can be provided.

Since replay takes place at the machine code level the amount of information required is usually large. All inputs and intermediate events, e.g. messages, must be kept.

Reply: The amount and the necessary information required is of course a design issue, and it is not true that all inputs and intermediate messages must be recorded. The replay can as we have shown actually re-execute the tasks in the recorded event history. Only those inputs and messages which are not re-calculated, or re-sent, during the replay must be kept. This is specifically the case for RTS with periodic tasks, where we can make use of the knowledge of the schedule (precedence relations) and the duration before the schedule repeats it self (the LCM – the Least Common Multiple of the task period times.) In systems where deterministic replay has previously been employed, e.g., distributed systems [11] and concurrent programming (ADA) [14] this has not been the case. The restrictions, and predictability, inherent to scheduled RTS do therefore give us the great advantage of only recording the data that is not recalculated during replay.

Statement: If a program has been modified (e.g., corrected) there are no guarantees that the old event history is still valid.

If a program has been modified, the relative timing between racing tasks can change and thus the recorded history will not be valid. The timing differences can stem from a changed data flow, or that the actual execution time of the modified task has changed. In such cases it is likely that a new recording must be made. However, the probability of actually recording the sequence of events that pertain to the modification may be very low. As explained earlier, debugging in general and deterministic replay especially is not suited for speculative investigations of the system behavior. This is an issue for regression testing, as explained in [15][16].

7 Related work

There are a few descriptions of deterministic replay mechanisms (related to real-time systems) in the literature:

- A deterministic replay method for concurrent Ada programs is presented by Tai et al [14]. They log the synchronization sequence (rendezvous) for a concurrent program *P* with input *X*. The source code is then modified to facilitate replay; forcing certain rendezvous so that *P* follows the same synchronization sequence for *X*. This approach can reproduce the synchronization orderings for concurrent Ada programs, but not the duration between significant events, because the enforcement (changing the code) of specific synchronization sequences introduces gross temporal probe-effects. The replay scheme is thus not suited for real-time systems, neither are issues like unwanted side-effects caused by preempting tasks considered. The granularity of the enforced rendezvous does not allow preemptions, or interrupts for that matter, to be replayed. It is unclear how the method can be extended to handle interrupts, and how it can be used in a distributed environment.
- Tsai et al present a hardware monitoring and replay mechanism for real-time uniprocessors [17]. Their approach can replay significant events with respect to order, access to time, and asynchronous interrupts. The motivation for the hardware monitoring mechanism is to minimize the probe-effect, and thus make it suitable for real-time systems. Although it does minimize the probe-effect, its overhead is not predictable, because their dual monitoring processing unit causes unpredictable interference on the target system by generating an interrupt for every event monitored [1]. They also record excessive details of the target processors execution, e.g., a 6 byte immediate AND instruction on a Motorola 68000 processor generates 265 bytes of recorded data. Their approach can reproduce asynchronous interrupts only if the target CPU has a dedicated hardware instruction counter. The used hardware approach is inherently target specific, and hard to adapt to other systems. The system is designed for single processor systems and has no support for distributed real-time systems.
- The software-based approach *HMON* [1] is designed for the HARTS distributed (real-time) system multiprocessor architecture [13]. A general-purpose processor is dedicated to monitoring on each multiprocessor. The monitor can observe the target processors via shared memory. The target systems software is instrumented with monitoring routines, by means of modifying system service calls, interrupt service routines, and making use of a feature in the pSOS real-time kernel for monitoring task-switches. Shared variable references can also be monitored, as can programmer defined application specific events. The recorded events can then be replayed off-line in a debugger. In contrast to the hardware supported instruction counter as used by Tsai et al., they make use of a software based instructions counter, as introduced by [10]. In conjunction with the program counter, the software instruction counter can be used to reproduce interrupt interferences on the tasks. The paper does not elaborate on this issue. Using the recorded event history, off-line debugging can be performed while still having interrupts and task switches occurring at the same machine code instruction as during run-time. Interrupt occurrences are guaranteed off-line by inserting trap instructions at

the recorded program counter value. The paper lacks information on how they achieve a consistent global state, i.e., how the recorded events on different nodes can consistently be related to each other. As they claim that their approach is suitable for distributed real-time systems, the lack of a discussion concerning global time, clock synchronization, and the ordering of events, diminish an otherwise interesting approach. Their basic assumption about having a distributed system consisting of multiprocessor nodes makes their *software* approach less general. In fact, it makes it a hardware approach, because their target architecture is a shared memory multiprocessor, and their basic assumptions of non-interference are based on this shared memory and thus not applicable to distributed uniprocessors.

8 Conclusions

Traditional debugging methods provide little insight into the cause of a real-time system crash. The TimeMACHINE records execution and data history, and lets you re-execute and analyze the events leading up to a crash. The TimeMACHINE allow you to replay the history over and over again; forward, backward, and in greater detail than recorded by means of an instruction level simulator debugger or standard JTAG/BDM or ICE debuggers.

Since the histories (task-switches, interrupts, inputs, and communication) are replayed in the debugger, which has its own timeline, we can replay (re execute) the system history faster or slower than the recorded “executed real-time” (this is specifically the case with instruction level simulators). This also allows you to insert any number of breakpoints, single-step, probe or poke the system while still executing the exact same series of events that occurred during run-time.

There exists now a commercial tool based on the TimeMACHINE technology provided by ZealCore Embedded Solutions AB (www.Real-TimeMachine.com).

9 References

[1] Dodd P. S., Ravishankar C. V. *Monitoring and debugging distributed real-time programs*. Software-practice and experience. Vol. 22(10): 863-877, October 1992.

[2] Eriksson C., Thane H. and Gustafsson M. *A Communication Protocol for Hard and Soft Real-Time Systems.* In the proceedings of the 8th Euromicro Real-Time Workshop, L'Aquila Italy, June, 1996.

[3] Gait J. *A Probe Effect in Concurrent Programs*. Software – Practice and Experience, 16(3):225-233, March, 1986.

[4] Kopetz H and Ochsenreiter W. *Clock Synchronisation in Distributed Real-Time Systems*. IEEE Transactions on Computers, August 1987.

[5] Kopetz H. *Sparse time versus dense time in distributed real-time systems*. In the proceedings of the 12th International Conference on Distributed Computing Systems, pp. 460-467, 1992.

[6] Kopetz, H. and Kim, K. *Real-time temporal uncertainties in interactions among real-time objects.* Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, AL, 1990.

[7] Lamport L. *Time, clock, and the ordering of events in a distributed systems*. Communications of ACM, (21):558-565: July 1978.

[8] LeBlanc T. J. and Mellor-Crummey J. M. *Debugging parallel programs with instant replay*. IEEE Transactions on Computers,36(4):471-482, April 1987.

[9] McDowell C.E. and Hembold D.P. *Debugging concurrent programs*. ACM Computing Surveys, 21(4):593-622, December 1989.

[10] Mellor-Crummey J. M. and LeBlanc T. J. *A software instruction counter*. In Proc. of 3d International Conference on Architectural Support for Programming Languages and Operating Systerns, Boston, pp. 78-86, April 1989.

[11] Netzer R.H.B. and Xu Y. *Replaying Distributed Programs Without Message Logging.* In proc. 6th IEEE Int. Symposium on High Performance Distributed Computing. Pp. 137-147. August 1997.

[12] Schütz W. *Fundamental Issues in Testing Distributed Real-Time Systems*. Real-Time Systems journal, Kluwer A.P., vol. 7(2):129-157, 1994

[13] Shin K. G. *HARTS: A distributed real-time architecture.* IEEE Computer, 24(5):25-35, May, 1991.

[14] Tai K.C, Carver R.H., and Obaid E.E. *Debugging concurrent Ada programs by deterministic execution*. IEEE Transactions on Software Engineering. Vol. 17(1):45-63, January 1991.

[15] Thane H. and Hansson H. *Handling Interrupts in Testing of Distributed Real-Time Systems*. In proc. Real-Time Computing Systems and Applications conference (RTCSA’99), Hong Kong, December, 1999.

[16] Thane H. and Hansson H. *Towards Systematic Testing of Distributed Real-Time Systems*. Proc. 20th IEEE Real-Time Systems Symposium (RTSS’99), Phoenix, Arizona, December 1999.

[17] Thane H. and Hansson H. *Using Deterministic Replay for Debugging of Distributed Real-Time Systems*. In 12th EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS, pages 265-272 Stockholm , June 2000. IEEE Computer Society.

[18] Tsai J.P., Fang K.-Y., Chen H.-Y., and Bi Y.-D. *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging.* IEEE Transactions on Software Engineering, 16: 897 - 916, 1990.