Using Deterministic Replay for Debugging of Distributed Real-Time Systems

Henrik Thane and Hans Hansson
Mälardalen Real-Time Research Centre, Department of Computer Engineering
Mälardalen University, Västerås, Sweden, hte@mdh.se

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 a novel software-based approach for cyclic debugging of distributed real-time systems. 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.

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 distributed 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

In moving from debugging sequential non real-time programs to debugging sequential real-time programs temporal constraints on interactions with the external process have to be met. This means that classical debugging with breakpoints and single-stepping cannot be directly applied since it would make timely reproduction of outputs to the external process impossible. Likewise, using a debugger we cannot directly reproduce inputs to the system that depend on the time when the program is executed, e.g., readings of sensors and the local real-time clock. 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 we must in addition have mechanisms for reproducing task interleavings. We need for example, to keep track of preemptions, interrupts, and accesses to critical regions. That is, we must have mechanisms for reproducing the interactions and synchronizations between the executing tasks.

Reproducing rendezvous between tasks have been covered by Tai et al. [14], as have reproduction of interrupts and task-switches using special hardware, Tsai et al. [17]. Reproducing interrupts and task switches using both special hardware and software has been covered by Dodd et al. [1]. However, since both the two latter approaches are relying on special hardware and profiling tools they

are not very useful in practice. They also lack support for debugging of distributed real-time systems, even though Dodd et al. claim they do.

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 deterministic replay

We will in this paper present a software based debugging technique based on deterministic replay [8][9], which is a technique that records significant events at run-time and then uses the recording off-line to reproduce and examine the system behavior. The examinations can be of finer detail than the events recorded. For example, by recording the actual inputs to tasks we can off-line re-execute the tasks using a debugger and examine the internal behavior to a finer degree of detail than recorded.

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]. Probe-effects occur when the relative timing in the system is perturbed by observations, e.g., by breakpoints put there solely for facilitating observations. 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. If we record all interactions between the nodes we can locally on each node deterministically reproduce them and 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 entirely software based method for deterministic debugging of single tasking and multi-tasking real-time systems.*
- *The first method for deterministic debugging of distributed real-time systems.*

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, and *Section 6* gives an overview of related work. Finally, in *Section 7*, 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 a run-time real-time kernel that supports preemptive scheduling, and require that the kernel has a recording mechanism such that significant system events like task starts, preemptions, resumptions, terminations and access to the real-time clock can be recorded, as illustrated in Figure 1. The detail of the monitoring should penetrate to such a level that the exact occurrence of

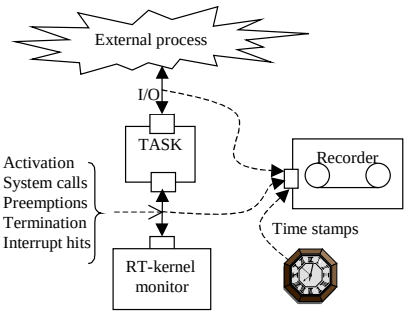

Figure 1 Kernel with monitoring and recording.

preemptions and interrupt interference can be determined, i.e., it should record the program counter values where the events occurred. All events should also be time-stamped according to the local real-time clock.

We further require that the recording mechanism governed by the run-time kernel supports programmer defined recordings. That is, there should be system calls for recording I/O operations, local state, access to the real-time clock, received messages, and access to shared data.

All these monitoring mechanisms, whether they reside in the real-time kernel or inside the tasks, will give rise to probe-effects [3][9] if they are removed from the target system. That is, removing the monitoring probes will affect the execution, thereby potentially leading to new and untested behaviors. The probes should therefore remain in the target system. It is consequently essential to consider monitoring early in the design process and allocate resources for it.

We further assume that we have an off-line version of the real-time kernel (shown in Figure 2), where the real-time clock and the scheduling have been disabled. The off-line kernel with support by a regular debugger is capable of replaying all significant system events recorded. This includes starting tasks in the order recorded, and making task-switches and repeating interrupt interference at the recorded program counter values. The replay scheme also reproduces accesses to the local clock, writing and reading of I/O, communications and accesses to shared data by providing recorded values.

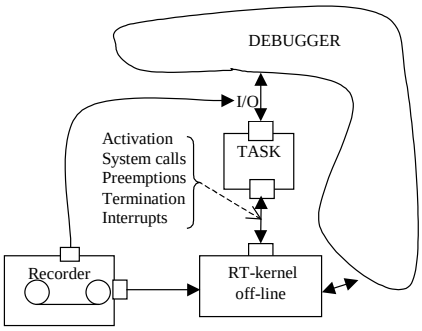

Figure 2 Offline kernel with debugger

3 Real-time systems debugging

We will now in further detail discuss and describe our method for achieving 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.

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.

However, if we identify and record significant events in the execution of the sequential program, such as reading values from an external process, accesses to the local clock, and outputs to external processes, we can order them. By ordering all events according to the local clock, and recording the content of the events (e.g., the values read) together with the time when they occurred we can off-line reproduce them in a timely fashion. That is, during debugging we "short-circuit" all events corresponding to the ones recorded by substituting readings of actual values with recorded values.

An alternative to our approach is to use a simulator of the external process, and synchronize the time of the simulator with the debugged system. However, simulation is not required if we already have identified the outputs that caused the failure.

3.2 Debugging multitasking real-time systems

To debug multitasking real-time systems we need, in addition to what is recorded for single task real-time systems, to record task interleavings. That is, we should record the transfers of control. To identify the time and location of the transfers we must for each transferring event assign a time stamp, and record the program counter (PC).

To reproduce the run-time behavior during debugging we replace all inputs and outputs with recorded values, and instrument all tasks by inserting trap calls at the PC values where control transfers have been recorded. These trap calls then execute the off-line kernel, which has all the functionality of a real-time kernel, but all transfer of control, all accesses to critical regions, all releases of higher priority tasks, and all preemptions by interrupts are dictated by the recording. Inter-process communication is however handled as in the run-time kernel, since it can be deterministically reproduced by reexecuting the program code.

Figure 3 depicts a schedule with the three tasks *A*, *B*, and *C*. We can see that task *A* is being preempted by task *B* and *C*. During debugging this scenario can be reproduced by instrumenting task *A* with calls to the kernel at *A*'s PC=*x* and PC=*y*.

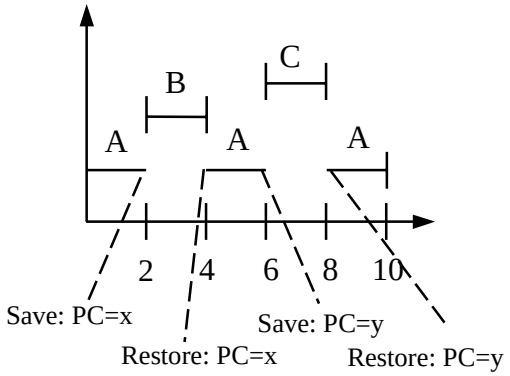

Figure 3 Task A is preempted twice by task B and C.

The above reasoning is a bit simplistic when we have program control structures like loops and recursive calls, since in such structures the same PC value will be executed several times, and hence the PC value does not define a unique program state. This can be alleviated if the target processor supports instruction or cycle counters. The PC will together with any of these counters define a unique state. However, since these hardware features are not very common in commercial embedded micro-controllers, we will use the following alternative approach:

Instead of just saving the PC, we will save all information stored by the kernel in context-switches, including CPU registers (address and data), as well as stack and program counter registers pertaining to the task that is preempted. The entire saved context can be used as a unique marker for the preempted program. The program counter and the contents of the stack register would for example be sufficient for differentiating between recursive calls.

For loops, this approach is not guaranteed to uniquely identify states, since (at least theoretically) a loop may leave the registers unchanged. However, for most realistic programs the context together with the PC will define a unique state. Anyhow, in the unlikely situation, during replay, of having the wrong iteration of a loop preempted due to indistinguishable contexts, the functional behavior of the replay will be different from the one recorded – and therefore detectable; or if the behaviors are identical, then it is of no consequence.

Any multitasking kernel must save the contexts of suspended tasks in order to resume them, and in the process of making the recording for replay we must store contexts an additional time. To eliminate this overhead we can make the kernel store and retrieve all contexts for suspended tasks from the recording instead, i.e., we need only store the contexts once.

Our approach eliminates the need for special hardware instruction counters since it requires no extra support other than a recording mechanism in the real-time kernel. If we nonetheless have a target processor with instruction counters or cycle counters, we can easily include these counters into the recorded contexts, and thus guarantee unique states.

To enable replay of the recorded event history we insert trap calls to the off-line kernel at all recorded PC values. During replay we consequently get plenty of calls to the kernel for recorded PC values that are within loops, but the kernel will not take any action for contexts that are different from the recorded one.

An alternative approach to keep track of loop executions is to make use of software instruction counters [10] that count backward branches and subroutine calls in the assembly code. However, this technique requires special target specific tools that scan through the assembly code and instrument all backward branches. 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.

3.3 Debugging distributed real-time systems

We will now show how local recordings can be used in achieving deterministic distributed replay. The basic idea is to correlate the local time stamps up to the precision of the clock synchronization. This will allow us to correlate the recordings on the different nodes. 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. Time stamps of all events make it possible to debug the entire distributed real-time system, and enables visualizations of all recorded and recalculated 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, or emulated on a single node for a set of homogenous nodes.

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?

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.

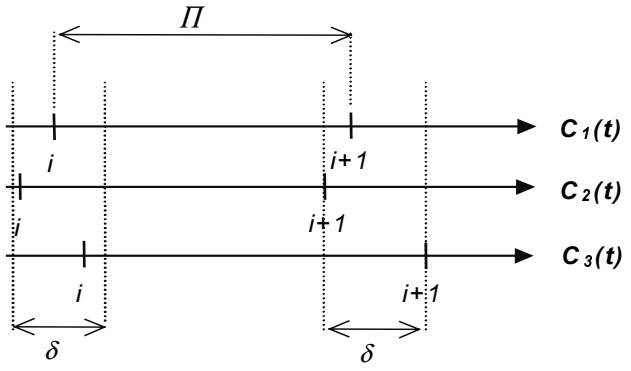

Figure 4 The occurrence of local ticks on three nodes

Figure 4 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].

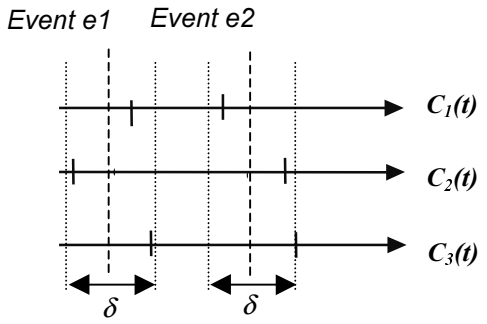

Figure 5. The effects of a sparse time base.

Consider Figure 5, 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 concurrent (i.e., having identical time stamps), while other nodes will assign distinct time stamps to the same events. This is illustrated in Figure 5, 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.

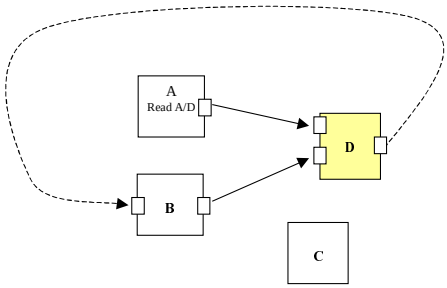

Figure 6 The data-flow between the tasks

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 6). The tasks *A*, *B*, and *D* are functionally related and 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 *D*, and task *D* receives both the processed A/D value and a message from *B*; subsequently *D* sends a new message to *B*.

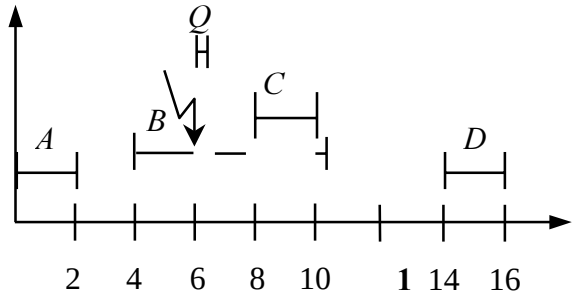

Figure 7 The recorded execution order scenario

Task *C* 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 7.

However, task *C* 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 *D*, 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 *C* 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 0 -16:

1. Task *A* starts at time 0
2. Task *A* stops at time 2
3. Task *B* starts at time 4
4. Interrupt *Q* starts at time 6, and preempts task *B* at PC=*x*
5. Interrupt *Q* stops at time 6,5
6. Task *B* resumes at time 6,5, at PC=*x*
7. Task *C* starts at time 8, and preempts task *B* at PC=*y*
8. Task *C* stops at time 10
9. Task *B* resumes at time 10, at PC=*y*
10. Task *B* stops at time 10,3
11. Task *D* starts at time 14
12. Task *D* stops at time 16

Together with the following data recording:

1. Task *A* at time 1, read_ad() = 234
2. Task *B* at time 4, message from *D* = 78

During debugging all tasks are instrumented with calls to the off-line kernel at their termination, and preempted tasks *B* and *C* are instrumented with calls to the off-line kernel at their recorded PC values. Task *A*'s access to the read_ad() function is short circuited and feed with the recorded value instead. Task *B* gets at its start a message from *D*, which is recorded before time 0.

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

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 Discussion

Schütz [12] has made three claims about deterministic replay in general, which we briefly comment below:

Claim 1: *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.*

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.

Claim 2: *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].

Claim 3: *The recording can only be replayed on the same hardware as the recording was made on.*

The event history can only be replayed on the target hardware. This is true to some extent, but should not be a problem if remote debugging is used. The replay could also be performed on the host computer if we have a hardware simulator, which could run the native instruction set of the target CPU. Another possibility would be to identify the actual high-level language statements where task switches or interrupts occurred, rather than try to replay the exact machine code instructions, which of course is machine dependent. In the latter case however, we run into the problem of defining a unique state when differentiating between iterations in loops.

6 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 minimizes 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.

7 Conclusions

We have presented a method for deterministic debugging of distributed real-time systems. The method relies on an instrumented kernel to on-line record the occurrences and timings of major system events. The recording can then, using a special debug kernel, be replayed off-line to faithfully reproduce the functional and temporal behavior of the recorded execution, while allowing standard debugging using break points etc. to be applied.

The cost for this dramatically increased debugging capability is the overhead induced by the kernel instrumentation and by instrumentation of the application code. To eliminate probe-effects, these instrumentations should remain in the deployed system. We are however convinced that this is a justifiable penalty for many applications.

We are currently implementing an experimental real-time kernel for evaluation of the presented debugging method, but also investigate modifications of existing real-time kernels to support deterministic replay.

8 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] 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.