Supervisor : Henrik Thane, Mälardalens University

Debugging the Asterix Framework by Deterministic Replay and Simulation

Daniel Sundmark, 750409
e-mail : it96das@student.docs.uu.se

Mälardalen Real-Time Research Centre, MRTC
Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN

Abstract

In recent years, the focus on embedded real-time systems has largely intensified. More and more products use embedded microcontrollers in order to enhance their functionality and user-interfaces. Unfortunately, analysis tools for real-time systems have not followed the same rapid development. In addition, analysis of real-time systems, like monitoring, debugging and visualization, is made difficult from the temporal and embedded nature of real-time systems. This paper describes a prototype of a real-time debugging system for the Asterix real-time kernel. The system proposed supports deterministic replay, a way of reproducing executions in order to isolate the bugs that cause errors. The executions are replayed in a debugger using a software simulator of the target hardware, making it possible to track down bugs only using a host PC for debuggging.

Available from Mälardalen Real-Time Research Center, Department of Computer Engineering, Mälardalen University, P.O. Box 883, SE-721 23 Västerås, SWEDEN

Contents

Chapter 1

Introduction

This chapter consists of five parts: First, a background to the research area of real-time systems analysis is given. Second, the main idea behind the Asterix framework is presented. This is followed by a description of the requirements put on the implementation of this thesis project, the Asterix kernel debugging system. Fourth, a brief summary of the thesis work is presented and fifth, the outline of the remaining document is presented.

1.1 Background

In recent years, the focus on embedded real-time systems has largely intensified. A diminishing percentage of all processors manufactured today are processors for personal computers. Instead, small microcontrollers, designed for embedded applications are dominating the market. These units are suitable for a wide range of systems including a vast number of applications. The main advantages of an embedded microcontroller are low power consumption, high stability and low cost. Compared to a dedicated regulating circuit, it also provides a greatly improved flexibility, due to the software implementation of the regulating system.

Together with a set of real-time tasks and often a real-time operating system, the microcontroller forms an embedded real-time system. A real-time system differs from an ordinary computer system in that the correctness of the system not only depends on the system's ability to produce correct results, but also on the system's ability to produce those results in a given time-interval. Real-time systems typically interact with an external process by sampling and responding to sampled data. One example of such a system is a system controlling the airbag releases in a car. When a collision occurs, the airbag must be activated. This is the functional correctness of the system. In addition, the airbag must not be activated too soon or too late as it would cause the driver to plunge into an not entirely filled airbag. This is the temporal correctness of the system.

Due to the embedded and the temporal nature of embedded real-time systems, the ability to efficiently monitor and observe the system behavior and activities has always been poor. I/O resources are sparse and adding basic system monitoring support, such as print-outs to a display, will affect the timing of the system. This can be lethal to real-time system temporal correctness. In addition, sufficient debugging support is most often not included in real-time system development environments. This makes the process of real-time software engineering an even harder one than that of normal software engineering.

So why do many existing real-time operating systems lack sufficient debugging support? This is a simple question, but the answer is quite complex. Traditionally, debugging is the process of revealing and isolating errors that cause already discovered failures. This is most often done by reproducing an erroneous execution in a specialized analysis tool called a debugger. To be certain to reproduce the same failure, the reexecution must follow exactly the same path through the execution as the initial run. In other words, we need to achieve a deterministic replay [7] of the initial execution. In their article, Thane and Hansson state that for ordinary sequential programs, providing the same start conditions and the same internal state is sufficient to achieve a deterministic replay of the initial execution. For multi-tasking real-time programs, several factors, such as interactions with an external process, task interleavings and accesses to the internal real-time clock, make the process of reaching an deterministic replay more difficult.

Most solutions proposed to these problems pay a high price, sometimes in efficiency, sometimes financially and not seldom in flexibility, due to hardware-dependent monitoring of kernel activities. These approaches will be discussed later in the *Basics* section of this paper.

1.2 The Asterix real-time kernel

Lack of sufficient debugging tools is not the only problem with existing real-time operating systems. Engberg and Petterson [3] claim that most current off-the-shelf RTOS are based on old assumptions of real-time systems and lack support for state-of-the-art scheduling and analysis theory and that this raises a problem when new theories are to be designed, implemented and tested. Some real-time operating systems, however, are based on state-of-the-art RTOS theory, but these are in most cases dedicated systems, specialized towards just one type of hardware or one type of scheduling theory. Furthermore, answers to exactly what features are supported in an existing RTOS can often be hard to find. In addition, although most commercial RTOS are configurable to fit a variety of embedded applications, not many provide the developer with the source code of the operating system, needed to make changes in the system.

All these drawbacks of existing RTOS gave birth to an idea of a new RTOS at the Department of Computer Engineering at Mälardalens University. The new system would consist of a highly configurable compiling distributed real-time kernel, a task model supporting state-of-the-art scheduling theory, wait- and lock-free interprocess communication and support for monitoring and debugging. All these features are collected in an open-source framework, called the Asterix framework [3] [2]. This paper is a description of the monitoring and debugging system of the Asterix kernel.

1.3 Requirements of the Asterix debugging system

Most of the requirements of the Asterix debugging system are extracted from the paper *Deterministic Replay for Debugging of Distributed Real-Time Systems* [7] by Henrik Thane and Hans Hansson. Some of the requirements are however specific to the Asterix kernel and are extracted from the papers dealing with the Asterix Framework [3] [2]. Below is a list of the most important requirements of the Asterix debugging system.

- The key requirement of the Asterix debugging system is that of determinism. The system must be able to reproduce an execution in such a way that all program states visited in the first run of the execution must be present in the replay execution, and in the same order.
- The system must be able to provide the developer with all the debugging possibilities as in an ordinary single-tasking non-real-time debugging session, unless the developer interacts with the debugging run in such a way that the flow of control is changed and the debugging run is corrupted.
- All jitter caused by the probes and the monitoring mechanisms must be minimized.
- The interference to the system done by the probes and the monitoring mechanisms in terms of CPU cycles must be minimized.
- The interference to the system done by the control flow and data flow recording buffers in terms of memory storage space must be minimized.
- All Asterix kernel features, such as signals, wait-free communication and semaphores must be supported by the Asterix debugging system.
- All probe and monitoring mechanism overhead must be predictable and computable.
- The Asterix debugging system must be compilable, portable and scalable.

1.4 Summary

In the spring term of 2000, two master thesis works of major importance to this thesis were conducted at the Real-Time System Design Laboratory at Mälardalens University. The first one was called *Asterix: A prototype of a small-sized real-time kernel* [3] and consisted of a well-documented real-time kernel prototype. The other thesis was called *Obelix development environment* [2] and consisted of an equally well-documented development environment for the Asterix kernel. These thesis works were the first two contributions to the Asterix framework. This is the third, *Debugging the Asterix kernel by deterministic replay* and consists of monitoring and debugging support for the Asterix kernel. The monitoring and debugging configuration is built in into the Obelix configuration system. This thesis was conducted at the fall term of 2000 during a 20 weeks period, also at the Real-Time System Design Laboratory at Mälardalens University.

1.5 Document Outline

Next, we will be looking at the terminology used in this document in order to avoid misunderstandings of the concepts of this thesis. This is followed by a description of the problems of debugging multitasking real-time systems and how other contributions in this area have approached these problems. After that, the different parts of the Asterix debugging system are described in detail. Eventually, future work and conclusions will be discussed.

Chapter 2

Name definitions in the Asterix debugging system

This chapter deals with name definitions of different parts of the Asterix debugging system. For more general name definitions in the Asterix framework, see [3].

Deterministic replay

Deterministic replay is the process of re-creating an execution in a way that it runs through exactly the same path as the initial execution did.

Control flow

The control flow is the turn of events in an execution run. It describes how taskswitches, interrupts, kernel invocations, preemptions and task starts and stops interact with each other along the time-axis of an execution.

Control flow buffer

In the Asterix debugging system, the control flow buffer is a memory storage space implemented as a cyclic buffer in order to record the control flow events of the current run.

Data flow probes

A data flow probe is the mechanism used in the Asterix debugging system to support the recording of significant user-task data, such as readings of sensors, received messages and task states.

Data flow buffers

The data flow buffers are a set of cyclic buffers, used to record the data monitored by the data flow probes.

Conditional breakpoint

A conditional breakpoint is a debugger breakpoint with a break condition. The breakpoint only halts the execution if the break condition is true.

Conditional breakpoint macro

A conditional breakpoint macro is a conditional breakpoint with a set of debugger commands associated with it. Once the breakpoint is activated, the commands are stepped through and executed one by one.

Chapter 3

Basics

```
task a {

   int x;

   while (x < 67) {
      .
      .
      .
      x = x + 1;
   }

}
```

Figure 3.1: Example of an uninitialized variable

While software engineers working with non-real-time software have been able to use debugging tools for decades, the real-time engineers have fought their battles without them. Some attempts to break this trend have been made, but the fact remains: A vast majority of all real-time systems today lack sufficient debugging support.

3.1 Defining the problem

So, what makes the process of debugging real-time systems so much harder than the one of debugging non-real-time systems? In the next three sections, we will in a step-by-step manner look at the differences of debugging non-real-time and real-time systems. A similar approach was made by Thane and Hansson [7].

3.1.1 Single-tasking non-real-time systems

We have already stated that debugging is the process of revealing and isolating bugs that cause errors discovered during run-time in a certain system. In a traditional single-tasking non-real-time program, this process is quite trivial. Consider, for example, the following lines of code in figure 3.1.

The code sequence will loop until the counter x reaches the value 67. However, the variable x is not initiated and for some compilers the code will work like the developer thought it would, but for other compilers, it might not. If this program would be run in a debugger, the developer would be able to reproduce an erroneous execution just by starting the program all over again with the exact same arguments (if any) as input. In the debugger, he would also be able to step through the program line by line or instruction by instruction and at each step examine whatever parts of the program state he would like to. In this example, watching the variable x would lead to an early discovery of the bug that led to the first erroneous execution.

3.1.2 Single-tasking real-time systems

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 motors, interacting with the context. In addition, the system is equipped with a real-time clock, giving the external and the internal process a shared timebase. 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 for achieving a deterministic replay. Second, how do we keep the shared timebase intact? In the debugging reproduction 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 run while the external process will continue. Consider, for instance, an ABS-breaking system in a car. During a testing phase, an error 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 timebase of the internal and external system is lost. This makes it impossible to reproduce the error deterministically and simultaneously thoroughly examining the state of the system at different times in the execution.

3.1.3 Multi-tasking real-time systems

The last step (of this thesis) is moving from single-tasking real-time systems to multi-tasking real-time systems. This adds the problem of concurrency within the system. 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 3.2, A beats B in a race situation for X and this leads to an error.

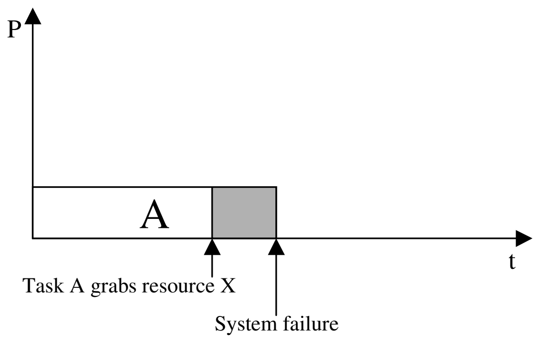

Figure 3.2: Task A grabs resource X before being preempted by task B.

The developer tries to investigate what led to the error and inserts some kind of software probe in the system in order to monitor what happened. When this probe executes, it causes a timing effect in the system which makes B beat A in the same race that A won in the first execution. This scenario is described in figure 3.3.

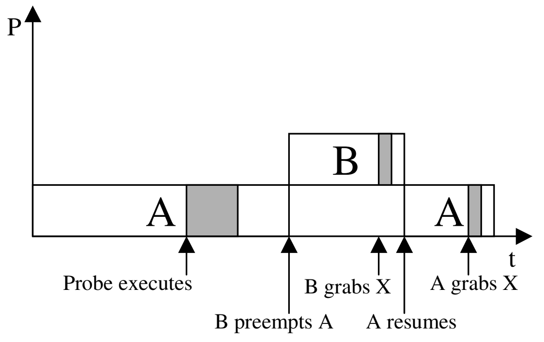

Figure 3.3: Task B preempts task A before A grabs resource X.

This time, the execution does not encounter any errors and the cause of the first error is still unknown. This type of behavior, when the insertion and removal of probes affect the execution of the system, is called *probe effects* [4].

3.2 Recent work

The research area of real-time system debugging is young and very few real-time targeted implementations of real-time debuggers have been presented. To enhance the description of previous work done in this area, one can expand the research area to the one of concurrent program debugging. Multi-tasking real-time debugging can be seen as a subset of concurrent program debugging and many of the difficulties appearing in concurrent program debugging will also appear in multi-tasking real-time systems.

In a survey studying the state-of-the-art of concurrent program debugging [6], McDowell and Helmbold define the problems of parallel program debugging:

> *The classic approach to debugging sequential programs involves repeatedly stopping the program during execution, examining the state, and then either continuing or reexecuting in order to stop at an earlier point in the execution. This style of debugging is called* ***cyclical debugging****. Unfortunately, parallel programs do not always have reproducible behavior. Even when they are run with the same inputs, their results can be radically different. These differences are caused by* ***races****, which occur whenever two activities are allowed to progress in parallel.*

Moving on from concurrent program debugging to real-time system debugging, we add the problems of temporal constraints on the system and interactions with an external process. However, we will continue with the ideas of McDowell and Helmbold as they make an attempt to divide the approaches of solving the problem of concurrent program debugging into four different subgroups. This is interesting since the same division can be made out of real-time debugging approaches. The first group is the one of traditional debugging techniques. The second one deals with event-based debuggers. The third describes techniques for displaying data and control flow in the system and the fourth deals with techniques for static analysis of data flow.

3.2.1 Traditional debugging techniques

Traditional debugging techniques can very well be used to find certain types of bugs in the code of real-time systems. Assuming one could isolate and debug the execution of every task in a multitasking system, all bugs not dependent on the timing of the system or the interaction with the external process can be found. Unfortunately, the most troublesome bugs are more malignant than that.

3.2.2 Event-based debuggers

The event-based debuggers use a different point of view when trying to debug multi-tasking real-time systems. In the example of the erroneous ABS-breaking system in section 3.1.2, we saw that an erroneous real-time execution can be virtually impossible to reproduce. Because of the fact that one never can be certain of achieving an identical reexecution of the one in which the error occured, the erroneous execution is recorded, analysed and reproduced in a debugger session. The recording can be done in two different ways: Either by using hardware-dependent noninterfering probes or by using software-dependent interfering probes. Both approaches have their own pros and cons.

Hardware-based solutions

Most, if not all, existing event-based debuggers use hardware-dependent probes to monitor the execution of the real-time system. Examples of such systems are proposed by Tsai et. al. [5] and Banda et. al. [1]. The main advantage of these systems is the ability to monitor sufficient information of the system without using any of the system's resources. The probes have different hardware-based techniques for gathering information, such as listening to the system bus, usually called *eavesdropping* or using *dual port memory*, that is memory circuits that can be read or written to by two systems simultaneously.

There are two major drawbacks of the hardware-based probes. First, the flexibility and portability is very low. If a new microcontroller is introduced to the market, new probe circuits need to be developed and distributed. For a hardware solution this is a time-consuming process. Second, the cost of such systems would be significantly higher due to production and distribution costs.

Software-based solutions

This is the section in which the Asterix debugging system could be included. The monitoring is performed by software mechanisms, leading to significantly increased flexibility and portability, but is at the same time intrusive. Intrusive probes may lead to increased system load or lack of reproduction correctness.

If the drawback would consist of decreased utilization or correctness depends on if the software probes are left in the system after the debugging phase is through or if they are removed. Removing the probes will lead to probe-effects in the system, which in turn might lead to new timing bugs. Letting the probes remain in the system will lead to decreased utilization of system resources, since CPU cycles and memory space are used to perform the recording.

Another example of a software-based solution is proposed by Mellor-Crummey et. al. who suggests a software instruction counter [] to make software-based real-time debugging easier.

3.2.3 Visualization techniques

When the control flow and data flow of the system is monitored, it can be used to visualize the execution history in different ways instead of building an replay basis. This can be useful for finding bugs, but will not provide the same possibility of examining the execution in depth as a cyclic debugging replay.

In addition, for event-based debuggers, like the Asterix debugging system, just building the replay basis does not remove the possibility of providing the user with a thorough visualization of the execution history. In the Asterix debugging system, a prototype text-based visualization is used to make the execution history easier to understand.

3.2.4 Formal methods

The last section is formal methods. These are the techniques that differ most from the traditional debugging techniques. There are a wide variety of formal method techniques, but most aim at describing the system specifications with some kind of formal mathematical language and to use this description to build and to verify the system.

The major reason for using formal methods is that the process of identifying and formalizing system requirements and specifications is very helpful in acchieving system design correctness. The drawbacks of formal methods include difficulty in understanding and learning the formal method and errors in translating system specifications to the formal mathematical language.

Chapter 4

The Asterix debugging system

In short, the Asterix debugging system is a set of programs and add-ons making it possible to record significant system events on-line, provide off-line possibility to analyze these data in order to build a complete basis for the deterministic replay mechanism. This chapter gives a description of the mechanisms needed to achieve a deterministic debugging in the Asterix Framework, starting with the monitoring process - how to collect data and monitor configuration. Next, the off-line analysis of monitored data is explained and last, the replay mechanism and the debugger interface are described.

4.1 Monitoring

Monitoring is an essential part of the Asterix debugging system. The monitor probes are the eyes and ears of the system and without them, interaction with the system would be impossible. However, vision and hearing are useless unless you know where to look and what to listen to.

4.1.1 What should be monitored?

To achieve deterministic replay of an execution, enough information of the execution must be gathered. Bare in mind though that, over-enthusiastic gathering of information will decrease overall system performance, since every monitored event or data cost time and memory. It is therefore important not to monitor and record too much data.

In order to make the Asterix debugging system easy to understand for developers and users, the storage space for monitored data is divided into two parts: Control flow buffers and the dataflow buffers.

4.1.2 The control flow buffer

The control flow buffer is used to store all significant kernel events, such as taskswitches, interrupts and missed deadlines. It consists of a cyclic buffer of a user-defined number of entries and an index pointing at the next entry available to write. Each entry in the buffer consists of a 7-tuple and is defined as:

$$event < T, ST, PC, SP, C, E, ET > \tag{4.1}$$

where T and ST define the time of the event. T is the value of an software-based tick counter, incremented by a timer interrupt. ST, on the other hand can be viewed upon as fractions of a tick and is the value of a free-running hardware counter, reset at each tick. This means that an event that occurred at T=43, ST=256 predates an event occurring at T=43 and ST=398. Henceforth, an event occurring at T=X and ST=Y will be referred to as occurring at $X.Y$. Moving on, we have PC, which is the value of the program counter register at the time of the event. SP is the value of the stack pointer and C is a 16-bit register checksum. Finally, we have E and ET. ET tells us what type of event that occurred and E is an identifier of the event. For example, if a hardware interrupt occurs, ET tells us that an interrupt occurred and E tells us which interrupt service routine that was run. Another example can be made out of a taskswitch. If a taskswitch occurred, ET tells us that a taskswitch occurred (and what type of taskswitch that occurred) and E tells us which task that gained control after the taskswitch. The structure of an event in the control flow buffer is shown in figure 4.1.

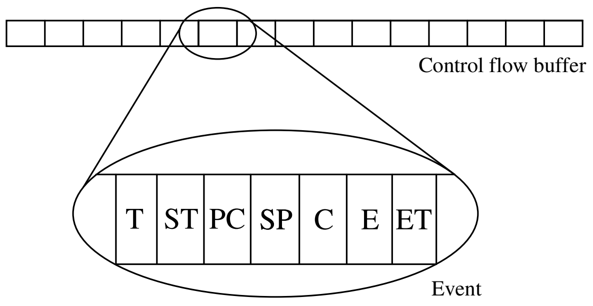

Figure 4.1: A magnified event in the control flow buffer.

In the Asterix kernel there are four different entries into the kernel, causing a taskswitch: *timertick*, *return*, *irq* and *yield*. The first one, *timertick* is called when a timer interrupt is generated. The second, *return*, is called each time a task terminates and wants to hand over the control to the kernel [1]. The third, *irq*, is a taskswitch generated by an external interrupt and the forth, *yield*, is a taskswitch generated by a semaphore

[1]All tasks in the Asterix Framework are of the terminating type.

race. In addition to these event types, *ET* can also indicate a missed deadline or, as mentioned above, an interrupt.

At the start of the execution, the control flow buffer is empty and the write index points to the first entry in the buffer. This is shown in figure 4.2.

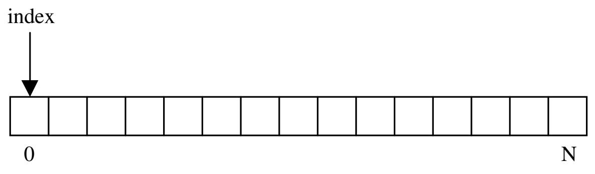

Figure 4.2: An empty control flow buffer of size N+1.

As an event occurs, the values of *T, ST, PC, SP, C, E* and *ET* at the time of the event are written to the first entry in the buffer. Then the write index is incremented, so that it points to the second entry in the buffer and so on. This is shown in figure 4.3.

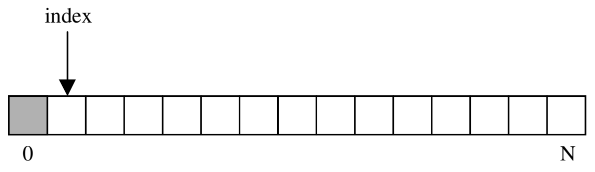

Figure 4.3: An event occurs, the entry is written to and the index is incremented.

When the index reaches the end of the control flow buffer, it is reset to its initial value. In other words, it points to the first entry of the buffer, which contains the oldest event. This can be seen in figure 4.4.

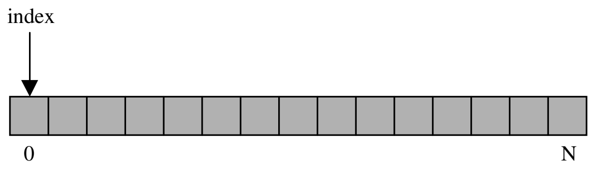

Figure 4.4: The index reaches the end of the buffer and starts over.

When an event occurs, the variables of the new interrupt overwrite the old ones. The old event is lost and will can not be reproduced during replay. It is therefor important that the developer in an early state of the design allocates enough space to achieve a replay long enough to reveal the causes of errors. That is, the precision of the replay is proportional to the amount of data recorded

4.1.3 The data flow buffers

The other part of the memory storage space for monitor data is the data flow buffers. This is a set of user-defined buffers, used to record user-task data instead of kernel events, which was the case with the control flow buffer. The data recorded here could be data external to the task, such as inputs to the task, system state, readings of sensors and messages from other tasks or nodes. In other words, data, that cannot be reproduced just by providing the same internal state as in the first run, should be recorded. A thing to keep in mind is that in contrast to the control flow buffer, each data flow buffer is tied to a specific user-task. The control flow buffer is global and is used only by the kernel.

An entry in a data flow buffer is defined as a 3-tuple:

$$data < T, ST, D > \tag{4.2}$$

where the meanings of *T* and *ST* are analogous to those in the control flow buffer. D, however, is a set of user-defined data recording entries, that can be used to record data during a run-time execution of the user-task. This might seem a little tricky, but consider the following example:

S is a system regulating the fluid level in an industry process. The system has three tasks: *A*, *B*, and *C*. *A* is the task reading the level of the fluid from two different sensors in the fluid tank. It then decides through a regulator if the flow into the tank should be decreased, increased or if it should remain constant. It sends this decision to task *B* as a message and task *B* translates the decision to motor control information for the motor controlling the fluid inflow hatch. Task *C* is an interrupt-driven task, independent from *A* and *B*, only responding to a button that resets the entire system.

So, how should the tasks in system *S* be monitored? Let us start with task *A*. We only need to monitor the data that is impossible to reproduce just by providing the same internal state and inputs to the system. The values of the level-sensor readings are such data. Believing that fluid flow is deterministically reproducible just by keeping the initial level close to the same is the same as saying that a dice will always show the same value if you try to toss it in the same velocity from the same level of height at different points in time. Reality is chaotic and minimal changes in the initial state of a process will show large effects on the outcome. Anyway, the two values read from the level-sensors each time task *A* is invoked must be recorded. Moving on to task *B* we see that it reads a message from task *A*. This wouldn't be a problem if the replay started at the same point as the beginning of the original run. However, this is a system that might run for weeks, even months, before running into an erroneous state. This means that the monitored execution history would be enormous and never fit in the memory of an embedded system. In the Asterix debugging system we make use of cyclic buffers of user-defined length, allowing us only to replay a certain time-interval back in the execution history. This leaves us with the possible problem of starting the replay at a point in time where *A* already sent a message, but *B* hasn't received it yet, and in the replay run, *B* never will. This is illustrated in figure 4.5, where *t0* is a execution point with a message in transit mode and *t1* is a point where no messages are in transite mode.

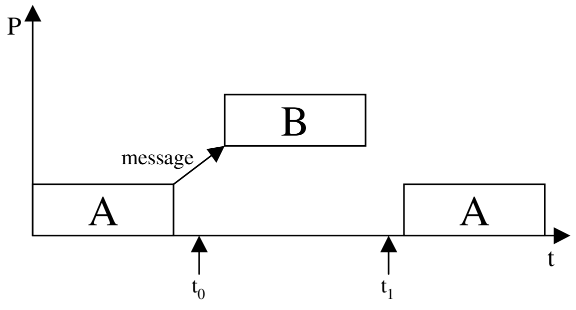

Figure 4.5: Transit mode.

There are two solutions to this problem: Either you can make sure that the replay starts at a *transitless state* [], a state where no messages are in transit-mode. Or you can record all messages received, making sure no messages are lost. Currently, the Asterix debugging system supports only the latter of these approaches. This means that task B in our system S needs to record the message from task A.

Looking at the last task in our system (task C), we see that the only external interaction here is the interrupt starting the task. But interrupts are covered by the control flow buffer (which is automatically added and configured if the system uses monitoring) and task C doesn't need to monitor any data.

As shown in figure 4.6, our system S has three tasks, but only two with task monitors. All buffer sizes are user defined, such as the types and names of the data recorded.

The data flow buffers are cyclic and their recording mechanism works in the exact same way as the control flow buffer.

4.2 Monitor configuration

The configuration of the Asterix debugging system is built into the Obelix configuration system [2] used to configure the Asterix kernel. Here, the developer has the ability to tell the system what size the monitor recording storage should be and what types and names the variables recorded should have. The easiest way of allocating space for the monitoring is to simply tell the system the size of the time-interval the recording should cover. If, for example, the developer tells the system to keep track of all events and data for the last 300 clock ticks, the Obelix configuration system translates this time-interval to control flow buffer and data flow buffer storage space.

```
Task_A_Monitor:
buffer of  < T, ST, < sensor1, sensor2>>

Task_B_Monitor:
buffer of < T, ST, < message >>

Task_C_Monitor:
null
```

Figure 4.6: Data flow buffers of system S

Doing this time-interval-storage-space translation for the data flow buffers of periodic user tasks is fairly trivial. If we have a periodic task, invoked at every 30 ticks and a recording time interval of 300 ticks, we need a data flow buffer size of $300/30 = 10$ to be absolutely certain that we have recorded all invocations of the task during the last 300 ticks.

Doing the same translation for the control flow buffer is a little more tricky. If we just consider recording taskswitches, we have a set of tasks with different priorities. All these tasks, but one (the task with the lowest priority, in the Asterix kernel, the idletask) has the potential to preempt another task's execution. This preemption will produce two taskswitches, first the one to the task with the higher priority, then the one back to the preempted task when the task with the higher priority terminates. This means that the maximum number of taskswitches N during a period of T ticks in a system of only periodic tasks can be calculated by the formula:

$$N = 2\sum_k T/PERk \tag{4.3}$$

Adding aperiodic task starts, interrupts and deadline misses, the algorithm becomes slightly more complex. A more thorough and user-oriented description of the configuration can be found in appendix A along with a monitor configuration tutorial.

4.3 Off-line analysis

The main objective of the off-line analysis is to provide a complete basis for the deterministic replay debugging mechanism. The input to the off-line analysis is the raw data of the control flow buffer and the data flow buffers recorded on the target system. The idea is to use the information from system events to create breakpoints in the replay execution where the events occurred in the recorded execution. Once halted at a breakpoint, different parts of the system state can be altered in order to simulate an interrupt or a taskswitch. When the system state variables are set, the execution resumes. In this way, all external system events can be simulated.

4.3.1 The debugger

The debugger used in the Asterix debugging system is the GNU source-level C-debugger, gdb. The current version of Asterix uses gdb-4.17 configured with a

```
break [BREAKPOINT] if [BREAK_CONDITION]
   commands
   [COMMAND_1]
   [COMMAND_2]
   .
   .
   [COMMAND_N]
   end
```

Figure 4.7: Structure of conditional breakpoint

```
break *0x8845 if (sp==0x91ae &&
               register_checksum==0x53c2)
```

Figure 4.8: Conditional breakpoint

h8300-hitachi-hms microcontroller software simulator. The simulator is used to perform the entire debugging on the host (or development) machine instead of using remote debugging with the h8300, which would decrease flexibility and debugging speed heavily.

Gdb supports script command files, which means that the breakpoints and settings of system variables can be defined as macros in script files, automatically generated by the Asterix off-line analysis tool. A gdb macro breakpoint can have different commands attached to it and will follow the syntax seen in figure 4.7.

Recent tests have shown that this approach should also work perfectly well with any commercial debugger supporting similar macros.

4.3.2 Analyzing control flow

As we stated earlier, a control flow buffer entry is made up of a 7-tuple: *event* $< T, ST, PC, SP, C, E, ET >$ and represents an event in the history of the recorded execution. To reproduce these events, each entry in the buffer has to be transformed into a conditional breakpoint macro. This is done in a number of steps. When the control flow buffer and the data flow buffers are uploaded from the recorded execution, they are represented as a bunch of unsorted raw data. Therefore, the first step is to separate the control flow buffer and the different data flow buffers from each other. Next, they are sorted in a chronological order. After that, each entry in the control flow buffer is translated into a gdb conditional breakpoint macro.

This conditional breakpoint will be slightly different for different types of events, but the principle is the same. Consider a fictional event, *e1:* $< T{=}12, ST{=}235, PC{=}0x8845, SP{=}0x91ae, C{=}0x53c2, E{=}3, ET{=}timertick >$. The first thing to do is to create the breakpoint and the breakpoint conditions. For *e1*, it would look something like the breakpoint in figure 4.8

The break condition is there to make sure that the execution is not halted at the wrong iteration of a loop or at the wrong instance of a recursive call, where the same program counter value is visited a number of times, but almost never with the same stack pointer or register checksum values. This is not a waterproof technique, but it is definitely enough to handle most situations.

```
break *0x8845 if (sp==0x91ae &&
               register_checksum==0x53c2)
   commands
   set tasklist[0].LDP = L0
   set tasklist[1].LDP = L1
   .
   .
   set tasklist[3].LDP = 0
   .
   .
   set tasklist[N].LDP = LN
   jump timertick
   end
```

Figure 4.9: Conditional breakpoint macro

To make sure that the system behaves in exactly the same way as in the recorded run, we must see to it that tasks are invoked at the right clock ticks. This is done by setting the task's *length-to-next-period (*LDP*)*-variable [3] at the different breakpoints. At event *e1*, the tick counter has reached a value of 12. We also see that event *e1* is a timertick taskswitch to task 3 (ET=timertick and E=3). In other words, task 3 is invoked at T=12 and the time distance to the next invocation of task 3 is 12 - 12 = 0 ticks. This means that the *LDP* variable of task 3 should be set to 0. This will cause the kernel to perform a taskswitch to task 3. In the conditional breakpoint macro, this will look like the one in figure 4.9.

Here, N is the number of tasks - 1 and Lx is the *LDP* calculated for task *x*. At each breakpoint, the *LDP* for all tasks is calculated and set. There are also a few additional commands in the conditional breakpoint macros, but most of these are static commands for making the debugging run a little smoother.

4.3.3 Analyzing data flow

Creating a replay basis of the data flow buffers is a lot easier than creating the conditional breakpoints from the control flow buffer. The data flow buffers are just a set of sorted data arrays, which are written back in the system by gdb command files. The data is written back at the same storage in memory where it once was recorded. Then the index is set to the initial value and when the replay debugging run starts, the same kernel mechanisms that once stored the data can be used to retrieve data instead. In other words, this time the data is read from the recording area into user tasks variables instead of the other way round.

4.3.4 Setting a valid replay starting point

The last step of the off-line analysis that has to be performed before we can start debugging the system by deterministic replay is the task of setting a valid replay starting point. Consider the recorded control flow in figure 4.10, representing an execution history.

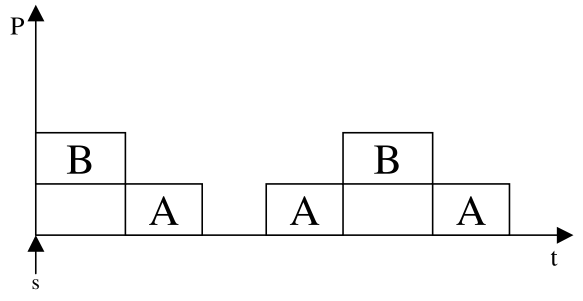

Figure 4.10: A non-valid replay starting point.

The first task executing at starting point *s* in our execution history is task B. After task B has terminated, task A is scheduled. However, we know nothing of this invocation of task A. Is it a resumed preempted execution or is it a initial task start? We don't know. The easiest way around this problem is to find the first point in the execution history where no user-tasks are in the ready-queue. We could call this an idle point. At such a point, no task will be in a preempted state and therefor all task executions will be reproducible. In figure 4.11, the first idle point in the recorded execution is identified and set to starting point *s*.

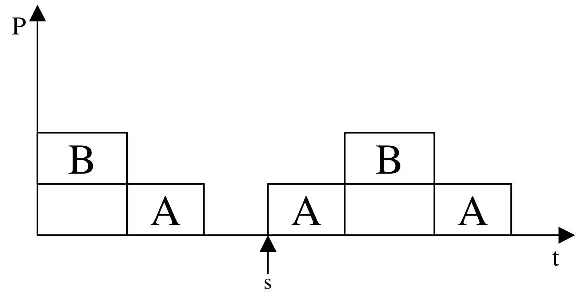

Figure 4.11: A valid replay starting point.

4.4 Replay mechanism

Having understood the concepts of the monitoring and the off-line analysis, there is little else to learn about the replay mechanism. Once all gdb command files are created by the Asterix off-line analysis tool, gdb can be started. A static initiation file is read into gdb, which in turn reads all dynamic command files created by the off-line analysis. This sets all conditional breakpoints in the system. After this, the system can be debugged in the same way as any ordinary non-real-time single-tasking system. The breakpoints causing the simulated events are stepped through and the replay execution will run exactly the same path as the recorded run did.

However, one has to remember that if the developer at some point in the debug execution interacts with the system in such a way that the flow of control of the system is altered, some breakpoint condition might never be met and the replay run is corrupted.

Chapter 5

Future work

Since this thesis is just a prototype of the Asterix debugging system, not all features of the system are optimized, thoroughly tested or even implemented. In this chapter, the most important improvements and new features are listed. Apart from those, there probably exist a few dozen that the author did not think of when writing this thesis.

5.1 Improvements

Due to limitations in time, some parts of this thesis was not implemented in the best possible way. Following is a set of system parts whose functionality and efficiency could be heavily improved.

5.1.1 Optimization

Although most software probes are well-optimized with regards to CPU cycles, there is work to be done in the area of jitter-minimization of probes. Furthermore, the size of the control flow and data flow recording buffers could be optimized to minimize the size of memory storage space needed to record an event or a data.

In addition, the control flow probes, monitoring the taskswitches and the hardware interrupts are quite messy and not very easy to understand just by looking at their code. They could very well be improved in a user-understandability point of view.

5.1.2 Name definitions

The variables of the Asterix debugging system source code don't follow any standardized variable name-patterns and provide the developer with little information of their function in the program.

Furthermore, some configuration variables in the revised Obelix grammar have confusing types and names. These variables should be altered to provide a more user-friendly configuration environment.

5.1.3 Calculating buffer sizes

The algorithm calculating the number of entries in the control flow buffer is not fully implemented and will currently set the buffer size to a pre-compilation declared integer. The algorithm itself, even when thoroughly implemented will only calculate a worst-case scenario size for the buffer, which in most cases might crave a much larger memory storage space than actually needed. An improvment of the algorithm would be welcome.

5.1.4 Off-line analysis robustness

Some parts of the off-line analysis tool, which reads recorded data and builds a basis for the debugging replay, are not well-enough tested. They will probably have some difficulties with certain types of recordings, especially in the process of setting a valid starting point for replay.

5.2 New features

The list of possible new features for the Asterix debugging system goes on and on, but there are a few that seem a little more important than the others.

5.2.1 Semaphores

The only thing that the Asterix debugging system lacks to be able to completely debug the entire Asterix kernel by deterministic replay is the support for semaphore recording and analysis. This will probably be the next feature added to the Asterix debugging system.

5.2.2 Distributed debugging

In the future, the Asterix Framework will support a network of real-time nodes, each running their own instance of the Asterix kernel. This will crave support for distributed debugging with a global timebase, allowing the system to debug the entire network.

Chapter 6

Conclusions

In the documentation of the Asterix kernel *kernel*, the authors discussed the possibility of adding monitoring and debugging mechanisms to the kernel in order to make the process of creating Asterix-based real-time systems easier than the one of most other real-time operating systems. In addition, debugging by deterministic replay support would make further development of the kernel easier and less frustrating.

In the same article, the authors also described the structures of the kernel that prepared the Asterix system for debugging by deterministic replay. The idea was to use the same mechanisms that save task states for the debugging recording. This would save us the trouble of saving the same data twice at each event. As it turned out, these structures were not used in the debugging recording due to their massive memory usage, but the changes were easily made and the basic idea remains the same.

In many articles promoting monitoring and debugging by hardware-dependent non-interfering probes, the software probe concept is said to be unacceptable due to two main reasons: First, the removal of the probes will cause probe-effects in the system, making it impossible to achieve an identical execution in the replay run or in the post-probe-removal runs. Second, the recording mechanisms and probes are said to use an unacceptably large percentage of system resources, such as CPU cycles and memory storage.

By this thesis, we have shown that multi-tasking real-time debugging by deterministic replay using software probes is no impossible-to-implement research project, but a working real-time analysis tool with acceptable losses in kernel performance and memory usage. The probe effect is eliminated by letting the monitoring probes remain in the system even after the debugging phase is through. Unfortunately, no thorough performance analysis of the monitoring mechanisms in the kernel have been made, so no exact numbers can be presented in this thesis.

The Asterix debugging system provides an easy-to-use interface and is flexible and easily portable due to its software implementation. Currently, the debugging system supports only one hardware platform, the Hitachi H8/300 microcontroller, used in the Lego Mindstorm platform, upon which the entire Asterix Framework prototype is built. In the near future, however, all platform-dependent parts of the Asterix Framework will be ported to other hardware platforms.

In conclusion, as were the case with the Asterix kernel prototype, not all requirements and visions have been implemented in a fully satisfactory way. However, there is a working prototype of the Asterix debugging system, which have been used for demonstration purposes and, hopefully, will be used in educational and real-life system development.

Appendix A

Configuring the Asterix debugging system - a tutorial

To help the developer to understand the process of configuring the Asterix debugging system, this tutorial is added to the thesis. In this tutorial, we will use the fictional system S described earlier in the thesis. Observe that the configuration of tasks is not explained in this tutorial. For task configuration, see the Obelix documentation *obelixconftool*. The entire configuration file of the system S is shown in the end of this tutorial.

S is a system regulating the fluid level in an industry process. The system has three tasks: A, B, and C. A is the task reading the level of the fluid from two different sensors in the fluid tank. It then decides through a regulator if the flow into the tank should be decreased, increased or if it should remain constant. It sends this decision to task B as a message and task B translates the decision to motor control information for the motor controlling the fluid inflow hatch. Task C is an interrupt-driven task, independent from A and B, only responding to a button that resets the entire system.

The first thing to do is to tell the system that debugging support should be included. This looks like this:

```
DEBUG {

};
```

This tells the system to activate the recording mechanisms in the kernel. However, no recording storage for control flow is allocated yet. Next, we decide that we would like to have a recording of at least the last 300 system ticks. This declaration would look like the following:

```
DEBUG {

   BUFFER_TIME = 300;
};
```

Ok, now storage space will be allocated for 300 ticks of control flow. However, no data flow buffers are included in the system. Task A will need a monitor to record the two data read from the level sensors and Task B will need a monitor to record the messege sent from A. The declarations of the data flow buffers are shown in the following lines.

```
DEBUG {

   BUFFER_TIME = 300;

   MONITOR PERIODIC A {
      DATAENTRY "sensor_t" sensor1;
      DATAENTRY "sensor_t" sensor2;
   };

   MONITOR PERIODIC B {
      DATAENTRY "message_t" message;
   };

};
```

This will allocate space for recordings of data from task A and B. The *sensor_t* and *message_t* declarations tell the system the types of the variables *sensor1*, *sensor2* and *message*. Last but not least, control flow space has to be allocated for aperiodic events, such as activation of task C. If we need to be able to store a maximum of two invocations of task C in 300 ticks, the declaration should look like the one in the next section.

```
DEBUG {

   BUFFER_TIME = 300;
   APER_EVENTS = 2;

   MONITOR PERIODIC A {
      DATAENTRY "sensor_t" sensor1;
      DATAENTRY "sensor_t" sensor2;
   };

   MONITOR PERIODIC B {
      DATAENTRY "message_t" message;
   };

};
```

If the task C also needed to record data, we would add a data flow buffer, looking like the one in figure A.1

```
MONITOR APERIODIC C {
   /* Records two invocations */
   NO_BUFFERS = 2;
   DATAENTRY "data_t" data;
};
```

Figure A.1: Aperiodic task monitor

```
/* X = user-defined integer */
SZ_CONTROLFLOW = X;
BUFFER_TIME = 300;
APER_EVENTS = 2;
```

Figure A.2: User-defined control flow size

and if we would like to configure the size of the control flow buffer ourselves instead of letting the system calculate it, we would add a line looking like the one in figure A.2.

In the Obelix configuration file, helpful comments are added to show users where to put the debug declarations.

The only thing left now to make the debugging support complete, is to describe how to use the data flow buffers in user code. Let us take a look at the sensor reading code of task A. This is shown in figure A.3.

To monitor these two sensor data, only four lines of code has to be inserted. The first two are the declaration and the definition of the data flow buffer used to monitor the data. The second and third are monitor calls, used to copy the data from the read sensor data to the data flow buffer storage. The calls are shown in figure A.4.

```
void a(void *ignore){

monitorAbuf_t* mon
   .

mon = (monitorAbuf_t*)getMonitor(self());
   .
   .
   .
sensorVal1 = readSensor1();
monitor(&(mon->sensor1), sizeof(mon->sensor1),
        &sensorVal1);
sensorVal2 = readSensor2();
monitor(&(mon->sensor2), sizeof(mon->sensor2),
        &sensorVal2);
   .
   .
   .
return;
}
```

Figure A.4: Task A monitoring code.

```
void a(void *ignore){

   .
   .
   .
sensorVal1 = readSensor1();
sensorVal2 = readSensor2();
   .
   .
   .
return;
}
```

Figure A.3: Task A sensor reading code.

Appendix A

Revised Obelix configuration language specification

This is the context free grammar for the Obelix configuration language. The grammar is a 4-tuple:
`(V, T, P, S)`

- `V` is syntactic variables, nonterminals, that denote sets of strings. The last two lines are the nonterminals added in this thesis.

```
V = { file, systemmode, ram, modes, mode, resolution, tasks, hardtasks,
      softtasks, task, activator, args, error_routine, resources,
      communication, waitfrees, waitfree, num_buf, readers, reader,
      synchronization, signals, signal, users, user, semaphores, semaphore,
      debug, monitor, monitors, dataentry, dataentries, no_buffers,
      sz_controlflow, buffer_time aper_events }.
```

- `T` is terminals, basic symbols from which strings are formed, The word "token" is a synonym for "terminal". Again, the last two lines are the tokens added in this thesis.

```
T = { SYSTEMMODE, SYSMODE, RAM, INT_CONST, MODE, ID, RESOLUTION,
      HARD_TASK, SOFT_TASK, ACTIVATOR, OFFSET, DEADLINE, PRIORITY,
      ROUTINE, ARGUMENTS, ERR_ROUTINE, STRING_CONST, WAITFREE, WRITER,
      TYPE, NUM_BUF, READER, SIGNAL, USER, SEMAPHORE, STACK,
      SZ_CONTROLFLOW, DEBUG, MONITOR, DATAENTRY, PERIODIC, APERIODIC,
      NO_BUFFERS, APER_EVENTS, BUFFER_TIME }.
```

- `P` is productions which specifies the manner in which the terminals and nonterminals can be combined to form strings. `P = { See figure A.1 }`. In addition to the old production rules, a new set of rules are incorporated into `P` to support configuration of the debugging system. These are presented in figure A.2

- `S` is the start symbol of the grammar.

```
S = { file }.
```

```
file            ->  systemmode ram modes
systemmode      ->  SYSTEMMODE = SYSMODE;
ram             ->  RAM = INT_CONST;
modes           ->  modes mode
                 |  mode
mode            ->  MODE ID { resolution tasks resources debug };
resolution      ->  RESOLUTION = INT_CONST;
tasks           ->  hardtasks softtasks
hardtasks       ->  hardtasks HARD_TASK task
                 |  epsilon
softtasks       ->  softtasks SOFT_TASK task
                 |  epsilon
task            ->  ID { ACTIVATOR = activator;
                         OFFSET = INT_CONST;
                         DEADLINE = INT_CONST;
                         PRIORITY = INT_CONST;
                         STACK = INT_CONST;
                         ROUTINE = ID;
                         args
                         error_routine };
activator       ->  INT_CONST
                 |  ID
args            ->  ARGUMENTS = STRING_CONST;
                 |  epsilon
error_routine   ->  ERR_ROUTINE = ID;
                 |  epsilon
resources       ->  communication synchronization
communication   ->  waitfrees
waitfrees       ->  waitfrees waitfree
                 |  epsilon
waitfree        ->  WAITFREE ID { WRITER = ID;
                                  readers
                                  num_buf
                                  TYPE = STRING_CONST; };
num_buf         ->  NUM_BUF = INT_CONST;
                 |  epsilon
readers         ->  readers reader
                 |  reader
reader          ->  READER = ID;
synchronization ->  signals semaphores
signals         ->  signals signal
                 |  epsilon
signal          ->  SIGNAL ID { users };
users           ->  users user
                 |  user
user            ->  USER = ID;
semaphores      ->  semaphores sempahore
                 |  epsilon
semaphore       ->  SEMAPHORE ID { users };
```

Figure A.1: Obelix configuration language

```
debug           ->  DEBUG { sz_controlflow
                            buffer_time
                            aper_events
                            monitors };
                 |  epsilon
sz_controlflow  ->  SZ_CONTROLFLOW = INT_CONST;
                 |  epsilon
buffer_time     ->  BUFFER_TIME = INT_CONST;
                 |  epsilon
aper_events     ->  APER_EVENTS = INT_CONST;
                 |  epsilon
monitors        ->  monitors monitor
                 |  epsilon
monitor         ->  MONITOR PERIODIC ID {
                    no_buffers
                    dataentries };
                 |  MONITOR APERIODIC ID {
                    no_buffers
                    dataentries };
no_buffers      ->  NO_BUFFERS = INT_CONST;
                 |  epsilon
dataentries     ->  dataentries dataentry
                 |  epsilon
dataentry       ->  DATAENTRY STRING_CONST ID;
```

Figure A.2: Obelix configuration language, debug extension

Bibliography

[1] V. P. Banda and R. A. Volz. Architectural support for debugging and monitoring real-time software. *Real Time, 1989. Proceedings., Euromicro Workshop on*, 1989.

[2] A. Davidsson and J. Lindgren. Asterix framework: Obelix development environment. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[3] A. Engberg and A. Petterson. Asterix: A prototype for a small-sized real-time kernel. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[4] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience 16, Mars 1986*, 1986.

[5] H. Chen J. J. P. Tsai, K. Fang and Y. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE transactions on software engineering, Vol. 16, No. 8, August 1990*, 1990.

[6] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys, Vol. 21, No. 4, December 1989*, 1989.

[7] H. Thane and H. Hansson. Deterministic replay for debugging of distributed real-time systems. *In proceedings of the 12th Euromicro Conference on real-time systems , Stockholm, June 2000.*, 2000.