Debugging Parallel Systems: A State of the Art Report

Joel Huselius
joel.huselius@mdh.se

September 2002

MRTC Report no. 63

ISSN 1404-3041 ISRN MDH-MRTC--x--SE

Debugging Parallel Systems: A State of the Art Report

Joel Huselius (joel.huselius@mdh.se)
Department of Computer Engineering
Mälardalens University, Västerås, Sweden

September 23, 2002

Abstract

In this State of the art Report (SotA), we will give an introduction to work presented in the area of debugging large software systems with modern hardware architectures. We will discuss techniques used for single- multi- and distributed systems. In addition we will provide pointers to work by large players in the field, and major conferences of importance.

We will discuss the debugging of parallel systems, these include systems that have complex software or hardware architectures. We will explain why distributed and multiprocessor systems as well as multitasking and/or real-time systems must be handled differently than less complex systems normally are during debugging. As we describe a general method for debugging parallel systems, we will also see that even other hardware and software architectures and devices will inflict upon the debugging process.

Contents

List of Figures

Chapter 1

Introduction

To debug a sequential software can be considered as fairly straightforward. All necessary tools are available, and it is only a matter of time until all bugs are removed. It is potentially possible to consider exhaustive testing of the system, that is to test every possible combination of inputs, to ensure that the system is correct.

However, if we consider parallel systems, the multitude of potential execution orderings increase dramatically from one per combination of input parameters in the sequential case, to millions or even more in the parallel case. It is then not feasible to perform exhaustive testing even if it would be possible, and the fact is that it is not always possible. Some inputs to the system may not be controllable, or even visible, resulting in that we can only hope that chance will help us test a sufficient amount of executions if we perform a reasonable number of tests. If we assume traditional techniques used for sequential systems, we can see that they are insufficient for the task of debugging and ensuring the correctness of the system. Even if we find a bug, it is not always given that we can derive the cause of the bug. Information about the cause is required in order to repair the system successfully.

One of the intentions of this report is to investigate the issues that makes debugging parallel systems so hard. We will describe the generally accepted method for debugging parallel systems trough the use of monitoring and execution reproduction, and describe the topics which are important to cover in an implementation of that method. In the context of monitoring, we discuss issues such as the probe effect which can change the behavior of a system. The observability problem which must be solved when monitoring distributed systems. And the probe-ability problem which states that the system must be open enough to allow monitoring.

Another intention of the report is to survey the possible errors that may occur in these systems. These include synchronization errors, race conditions, and timing related problems that primarily occur in real-time systems.

Furthermore, the report presents a survey of the state of art (work carried out in the academia) and state of practice (work carried out in the industry) in the field of debugging parallel systems. The focus here will be on state of the art as the industry (understandably enough) is reluctant to make detailed information about their products publicly available.

Finally, by using the rest of the report as background, we point out areas for our future research.

1.1 Outline

The outline of this report is as follows: Chapter 2 provide explanations and definitions for some fundamental terminology relevant to the rest of the report. Chapter 3 describe the categories of potential errors which may occur in parallel systems.

Thereafter, the two chapters that follow discuss the two basic activities in the generally accepted basic model for debugging parallel systems. Chapter 4 discuss how monitoring can be performed, which problems that are encountered, and discuss different approaches and tools found in the literature. Chapter 5 discuss issues which must be respected when constructing a replay-mechanism, and some different approaches found in the literature are also discussed.

Chapter 6 provides some ideas which are intended for our future work, the issues discussed have arisen during the work on this report. Finally, we provide a short summary of the report in Chapter 7.

Chapter 2

Terminology

In this section we will describe some fundamental issues regarding the problems that we investigate, and the type of systems that we assume.

In their article "Debugging Concurrent Programs" published in 1989 [30], McDowell and Helmbold refers to debugging as the process of locating, analyzing, and correcting suspected faults, the same definition is also found in an article by Tsai et al. [62, s5;p127]. Faults are referred to as the cause of violations to the system specification [30]. Schütz has similar opinions in his survey [48]. In this report we will survey the area of debugging parallel systems.

2.1 Tasks, Processes, and Threads

There are many names for the threads of control in a computer system, in this report we shall use the name "task", which is often used in the context of real-time systems. In other contexts, including some of our sources, the threads of control are called processes. In many cases, these are more or less the same, but real-time tasks normally have less complex code but more complex constraints. Examples of these constraints are deadlines, release-times, jitter, etc.

A real-time task may have a release-time which specifies the earliest point in time when the task is allowed to execute. The deadline of a task is the latest time at which the task is allowed to terminate. Should a task fail to complete before its specified deadline, is contribution to the computation cannot be considered usable. The severity of such a failure may be grave (for hard real-time systems), but there are systems (soft real-time systems) which are designed to allow some amount of misses.

Many real-time systems are of a periodic nature, for example sample-actuating loops, where a task is to be performed with a certain frequency. Note that two tasks in the same system may very well have different frequencies, and they also may be phase-shifted to each other.

Precedence orders are relations that constitute dependencies between events. For example, one must put on socks before shoes when dressing.

Because of these complex constraints it is a non-trivial task to perform the scheduling of such systems. Thus, scheduling of real-time systems has been an important research topic for more then two decades, and continue to be so.

Jitter between task instances is a consequence of the cooperative use of resources between tasks. For example, as the processing power must be shared, and different tasks may have different periodicities, scheduling of tasks will differ between task instances.

2.2 Faults, Errors, and Failures

Above, we used a definition to the term *faults* present in the literature, we will however comply to a slightly different definition recalled and refined by Thane [56, s3.2.1.1;p23]:

A failure is the non-performance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions [25, s9.1;p172]. That is, an input, X, to the component, O, yields an output, $O(X)$, non-compliant with the specification.

An error is a design flaw, or a deviation from a desired or intended state [25, s9.1;p172]. That is, if we view the program as a state machine, an error (bug) is an unwanted state. We can also view an error as a corrupted data state, caused by the execution of an error (bug) but also due to e.g., physical electromagnetic radiation.

A fault is the adjunged (hypothesized) cause for an error [21]. Generally a failure is a fault. but not vice versa, since a fault does not necessarily lead to a failure.

Or, in other words: A failure of an entity (system, component, function, etc.) is an observed violation to the specification of the entity. A failure is a fault in the output, or product, of the entity. An error is an unintended state in the entity. A fault is the cause of an error, it is the reason for its presence. If the propagation of the fault is not prevented, the fault will lead to an error.

Therefore, a failure of a programming team to write error-free software will lead to latent faults in the source code. Execution of these faults may lead to errors in the system state, which will in turn lead to failures if they are not prevented from contaminating the output of the system.

Also, if a component receives an invalid input (as a consequence of a failure in the supplier of the input), and fails to detect that the input is invalid, that is a fault. If the fault changes the systems state, that is an error. The error leads to a failure of the component operation if its presence is visible outside the component.

There are two ways of avoiding faults in a system [41, s1;p1]; *fault avoidance* i.e. to avoid the occurrence of faults in the system, and *fault tolerance* i.e. to provide correct output in spite of the occurrence of faults.

2.3 Fault Hypothesis

In order to find faults, some assumptions must be made to which faults that can occur in the system. We will later in this report review the different types of faults that may potentially arise in parallel systems, but we will in this section make a small example. Thane recalls in his Ph.D. thesis [56, s3.2.1.3;p27] that a system has a given *failure semantic* if the probability of that the system will experience types of failures (or *failure modes*) not covered by the failure semantic is sufficiently low. Further, Thane defines that a given *fault hypothesis* is the assumption that a system will comply to a certain failure semantic.

Byzantine faults [20] describe when faulty components continue to interact with their environment. The can then issue incorrect answers to questions, but do so in a fashion that does not alarm the receiver of the answer. The scenario may also have a "two-face" quality to it; a node that is experiencing a Byzantine fault may issue different answers to different instances of queries. Say that a faulty node answers a query about the todays special at the local restaurant, the correct answer would perhaps be "pancakes", but the faulty node may answer "fish". It is not possible for the requesting node to detect the incorrectness of the answer without checking the menu him self (or querying multiple nodes) as the answer lies with in the scope of the potentially valid answers. If we assume that we may experience Byzantine faults, we can never assume that a provided input is correct, and must therefore take extreme measures if we want to construct a system which will behave correctly. It is therefore important to, for a given system, define a fault hypothesis that is not overly pessimistic in order to keep the time required for development within acceptable boundaries.

2.4 Nondeterministic Programs

Kranzlmüller provides a definition of a nondeterministic program in his Ph.D. thesis [18, s4.2.4;p89] as follows:

> *"A program is nondeterministic, if - for a given input - there may be situations where an arbitrary programming statement is succeeded by one of two or more follow-up states. This freedom of choice may be determined by pure chance or unawareness of the complete state of the execution environment."*

Meaning that if one set of inputs may cause a task to, from one run to the next, behave differently, then the system is nondeterministic. Note that, according to this definition, a program is nondeterministic also if the irregularity of its products is completely depending on factors that are unknown but not necessarily unpredictable. Thus, a deterministic system can appear to be nondeterministic just because we lack the knowledge to understand it.

The opposite of a nondeterministic program or system, must clearly be a deterministic program. In the book "Communication and Concurrency" by Milner [32, s11.1;p233], the issue of determinism has been formally defined.

2.5 Parallel Systems

In our definition of parallel systems we incorporate both systems that are complex in their hardware architecture, and/or in their software architecture. Also, parallel systems may be either truly parallel, or concurrent (semi-parallel), concurrent systems being when a resource (for example a CPU) is more or less transparently shared in time between two or more tasks.

2.5.1 Hardware

Complex hardware can be heterogeneous or homogeneous, i.e. the nodes of the system are not necessarily uniform with respect to their hardware architecture. Many things, such as instruction sets, computation capabilities, and external resources may differ. The nodes can furthermore be distributed and/or multiprocessor systems with modern processor architectures and external devices such as hard disks. As different nodes in a distributed system have individually differing temporal propagations, timing is an interesting factor that complicates the process of getting a consistent ordering of the system events; the ordering of events is compromised as no global time-base exists. Advances in Very Large Scale Integration (VLSI) technology allows the construction of System-on-Chip (SoC) hardware. SoC-technology allows designers to place entire systems in one silicon chip, as this (among other things) allow reduced contact with off-chip components, that normally are slower, which increase performance. However, the reduction of off-chip information flows limits our visibility of the system - many of the hardware transactions between on-chip components may be invisible and uncontrollable [16].

In order to create a greater understanding for our term "complex hardware", we will in the next two sections describe issues of the SPARC processor architecture to exemplify the impact of co-processors and pipelines to the trap handling. We will focus on exceptions and interrupts, these can be triggered by external devices, intentionally by use of software code, or unintentionally by incorrect use of software code.

Floating-Point Instructions

In this section we give a short introduction to the handling of floating-point instructions in the SPARC architecture, the reason for the introduction shall become evident.

There are three different lengths of floating-point representation in the SPARC architecture, 32x32-bit single-precision, 32x64-bit double-precision, and 16x128-bit quad-precision registers. Some of these registers overlap, meaning that they cannot all be used simultaneously.

Unlike many other instructions, floating-point instructions are asynchronous. Simultaneously with the dispatching of an instruction, when the Program Counter (PC) of the Central Processing Unit (CPU) advances, the instruction is also executed, the results are visible and usable for subsequent instructions. Such is not the case when using floating-point instructions, these are queued for execution in the Floating-Point Unit (FPU), and a new instruction is fetched. Thus, the instruction may not even have begun its execution when a new is issued. If the floating-point instruction is followed by a couple of normal instructions, there may be quite a lot of instructions "in the pipe" at the time when the floating-point instruction is executed. If the instruction generates an exception, this will affect the rest of the instructions that have been issued after that the floating-point instruction was issued. This must be accounted for in the handling of the exception. Similar problems may also arise in the case of pipelined execution, should an instruction generate an exception late in the pipeline.

Trapping in the SPARC Architecture

According to Weaver and Germand [64], a trap is the action taken by the processor when it changes the instruction flow in response to the presence of an exception, interrupt, or `Tcc` instruction.

In this section, we describe interesting issues in the trapping functionality of the SPARC architecture. There are quite a lot of possible traps that may occur, in a file that has a path similar to `/usr/include/v9/sys/machtrap.h` we can find a list of the different traps possible. This list is machine specific. Note that some interrupts have allocated a larger space than others, this enables all of the trap routine to be situated in the trap table entry. Other interrupts must branch to free memory if they require more than five instructions, this may imply swapping and cache operations which will slowdown the execution of the trap handler.

Because of the nature of the invocation of traps, SPARC differentiates between four different categories of traps [64]; Precise, Deferred, Disrupting, and Reset traps. A type of trap belongs to one of these four categories.

Precise Traps Precise traps are results of the execution of a special instruction whose objective it is to raise the trap. This may be used in order to gain access to privileged instructions, in systems-calls or similar.

There are three conditions that must be true in the case of precise traps.

As the trap occurs, many registers including the PC and nPC register are saved, and execution is commenced at an address that have previously been defined for the type of trap that occurred. The nPC register points to the instruction that is to be executed directly after the completion of the instruction indicated by PC. In the case of precise traps, that saved PC register must point to the instruction that induced the trap into the system, and the saved nPC register must point to the instruction that is (was) to be executed immediately after that.

Furthermore, all instructions issued before the instruction that was the source of the occurred interrupt must have completed their execution.

Finally, the third condition is that all instructions that where intended to directly precede the instruction that was the source of the occurred interrupt must remain un-executed.

Deferred Traps Similar to the precise traps, the deferred traps are also induced by the execution of instructions, that is, they do not originate from external events. They may, however, originate from mismatch between the external environment and the assumptions made by software (e.g. bus-error). The difference between the two categories is that deferred traps allows the program state to be changed between the dispatching and the execution of the instruction (see Section 2.5.1 for an example).

If a deferred trap and a precise trap occurs simultaneously, with the exception of floating-point exceptions that may be deferred past precise traps, the deferred trap may not be deferred past the precise trap. The reason for that floating-point exceptions are a special case may be that they concern different parts of the CPU compared to those that may infer precise traps, and therefore one may assume a more relaxed policy in these cases. Also, the deferred trap must occur before any subsequent instruction attempts to use any modified register or resource that the trap inducing instruction used.

Disrupting Traps A disrupting trap originates from the assertion of an hardware interrupt, either triggered by external stimulus, or software execution.

In the case of software originated disrupting traps, these may be deferred. The difference between deferred traps, and deferred software originated disrupting traps is that the cause of the latter may lead to irrecoverable errors.

Reset Traps Reset Traps differ from disrupting traps in that execution of the running program is not resumed.

Discussion

As we have seen an example of, modern computer architectures are not trivial. Therefore will the tasks that are executing on machines that implement such architectures be harder to debug. In order to fully understand the execution of a task, every aspect of its execution must be considered.

It can be debated whether it is really feasible to acknowledge every detail of the architecture in order to find bugs in a system. Such may not be the case, but it is very important to keep in mind that every abstraction, every divergence from the real target, will make the debugging tool more blunt.

2.5.2 Software

Complex software could be multitasking applications with substantial inter-communication, note that (similarly with the hardware aspect above) nodes in a distributed system can also be heterogeneous with respect to their operating systems and task-sets. In systems that do not use strong synchronization between tasks, interactions are difficult to understand and predict off-line, and recreation of a certain execution order is not necessarily feasible as no information is available off-line that can determine that two executions are equivalent and it is therefore not possible to determine if the recreation of an execution has succeeded. Furthermore, the systems can also be composed by several components that may be off-the-shelf (also known as COTS). As the use of COTS limits the developers detailed understanding of the software functionality and do not allow modification to source code, debugging these systems can be quite cumbersome.

Complex systems may also have additional real-time constraints that must be fulfilled. The system may have as objective to monitor or control an external process and must therefore comply to rules inherent in the context of that external process. These constraints are typically modeled as deadlines, periodicities etc. of individual tasks, or sets of tasks, in the system.

Also visualizing executions in these types of systems is quite difficult. As the complexity of the system grows, more information is required in order to understand what is happening. Reducing that information to a minimum, displaying it in an easy to use, and easy-to-understand manner, is an important task.

Debugging these types of systems described above is still very much handicraft, and there are not many tools available that assist programmers in these tasks. Our long-term objectives is to remedy that.

In this report we explain general problems in debugging software, and also explain which other problems arise when software and hardware architectures are more complex. We also survey the previous work in the field of software debugging, both from the academia and the industry, with the focus on parallel systems.

2.6 Debugging Parallel Systems

In this section we will first describe how sequential programs are normally debugged, and give an introduction to why making use of this approach without modifications is unfeasible in real-time systems and many parallel systems. Thereafter we provide a brief outline to the basic idea of a how to facilitate the use of the normal debugging technique also in parallel systems which may even have real-time constraints.

2.6.1 Cyclic Debugging

The normal way of debugging software systems is to repeatedly use for example a debugger that has facilities like stepping, break pointing, and monitoring of individual variables. Also other methods, like printing program traces to a screen or file, are common. A program can be run repeatedly, in order for the programmer to narrow down his/hers search for the suspected error. This process is normally referred to as *Cyclic(al) Debugging* [30, 22], and is an efficient approach for single-node systems that has only one thread of execution. Under certain circumstances, also concurrent tasks may be efficiently debugged this way. Assumptions made are that experiments are interactive as well as repeatable, and that the programmer can monitor all relevant program information during program execution. If one or more of these assumptions are not meet, the approach will not have as good possibility of success as otherwise, but may be more or less applicable anyway.

The cyclic debugging strategy introduces an overhead to the system during the debugging activity. In systems where one or more tasks have temporal restrictions on their execution that will result in abnormal behavior if violated, this strategy has limited applicability. Also systems that have race conditions for system resources between system entities will behave in a way that differs from the normal execution. Examples of where such race conditions may occur are operating system scheduling and unsynchronized communication.

There is also another problem with cyclic debugging applied to distributed systems, which is that all nodes must have a coordinated behavior during the debugging phase [30]. As the program execution encounters a breakpoint, it is supposed to stop its execution, but this would be impossible to communicate to the other nodes of the system without any latencies. Therefore, nodes that would normally not be able to complete a certain workload at a certain time relative to another node, will be able to do so because the other node is stopped for an arbitrarily long time. Thus, breakpoints in distributed systems can cause the system to behave in a way that it would not, had the breakpoint not been present.

2.6.2 Monitoring and Execution Reproduction

As hinted in the above section, in order to debug real-time- and parallel systems, we must uncouple the propagation of time from the propagation of the system that we wish to debug. The literature suggests that this can be accomplished by first monitor, or eavesdrop on, one execution of the system that is to be debugged to such a level of detail that we can then reproduce that particular execution over and over again in some form of model of the system. What has been accomplished by that process is that the particular instance of the system can be debugged by means of cyclic debugging. By iterating the process, we can find and debug as many bugs as there is time for.

The process of monitoring systems will be discussed in detail in section 4, where we discuss different approaches and provide some information on related work. We will in the remainder of this report refer to that execution that is subject to monitoring as the *reference* execution. In section 5 we survey different methods for, by using monitoring output, reproducing an instance of a system.

We will, later in this report, provide a more thorough survey of the possible techniques to perform monitoring and execution reproduction.

Chapter 3

Errors in Parallel Systems

Sequential programs can have all the normal programming errors like unintended handling of pointers and mixing of variables, and also various syntax errors. These errors can be found during compile time, or by cyclic debugging or similar.

Clarke and McDermid provides a classification of different software errors [5]:

Control errors are those that force the task through another path than intended.

Value errors may be the assignment of incorrect values to the correct variable.

Addressing errors assign values to incorrect variables.

Termination errors are in some way related to control errors, but could concern failure to terminate a loop.

Input errors could be unintended input values from sensors, or erroneous parameterization.

But also other errors are possible, memory leakage for instance may have many causes: One is a control error which leads to failure to execute the `free()` function when intended, which may lead to loss of memory. Another is the absence of code, the call to the `free()` function may be absent in the code.

In addition to those errors that occur in sequential programs, the nature of parallel, distributed, and/or multitasking systems give rise to classes of errors that are not visible in sequential systems. Kranzlmüller summarizes in [18, s4.2.3;p87] that deadlocks and livelocks are common classes of errors in these systems. In addition, also problems related to race conditions in the system are possible [34, 38]. Thane [56] also states that interleaving related errors, and precedence violations are possible. Finally, in real-time systems, also timing errors are possible. We will in this section explain the above mentioned errors.

The motivation for this chapter is to provide a well motivated understanding for the inherent complexity of parallel systems. A fully fledged debugging system must respect at least every issue discussed in this chapter.

3.1 Errors of Synchronization

In this section, we will discuss three different types of errors, first interleaving errors, then deadlocks, and finally livelocks. Both livelocks [51, s5.2;p211] and deadlocks [6] can be considered as very well known phenomenon's, but we provide a short description here.

3.1.1 Interleaving Errors

In order to experience livelocks or deadlocks, the system must use some form of synchronization primitives. The use of such primitives is often well motivated and the use fills a well needed function, if they are not used to a sufficient degree the system may experience interleaving errors.

In semi-parallel systems, as tasks compete for execution resources, small slots of execution time are distributed to those that require it. This distribution is done in a fashion that does not allow, and should not allow, the individual tasks to know how its program propagation will be with respect to other tasks. Therefore, the use of shared resources must be protected by synchronization primitives, so that mutual exclusion is guaranteed. If this is not performed correctly, a task that uses a shared resource may be, unknowingly, interrupted by another task that also makes use of the resource.

Such misuse of resources may lead to many other errors of which two are data-inconsistency and erroneous pointer referencing.

3.1.2 Deadlock

As we have seen, synchronization primitives are required in parallel systems. However, the well known system deadlock may be the result of incautious resource management if there are several shared resources to go about.

Imagine the following chain of events (see Figure 3.1): A task T_A tries to lock the semaphore of shared resource S_1. T_A is then interrupted by task T_B which locks the semaphore associated with resource S_2 followed by an attempt to lock the semaphore of resource S_1. T_B will then stall, as that semaphore belongs to T_A, thus allowing T_A to continue its execution. Task T_A will then try to lock the semaphore of resources S_2, but will be blocked because task T_B already owns that semaphore.

Since neither T_A nor T_B can continue there execution beyond this point, this would result in a deadlock between T_A and T_B.

It was stated by Coffman et al. in the 1971 article “System Deadlocks” [6], that

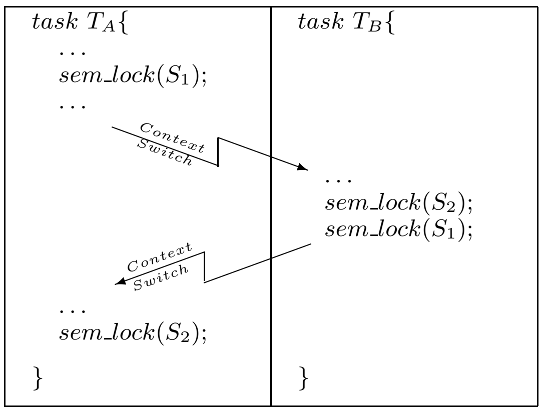

Figure 3.1: Example of a Deadlock

four conditions must be satisfied in order for a system to experience a deadlock:

Mutual exclusion: Tasks claims exclusive control of the (shared) resources they require.

Hold and wait: Tasks hold resources already allocated to them while waiting for additional resources.

No preemption: Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion.

Circular wait: A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain.

The circular wait condition implies that there are probably more then one process in the system, and probably more then one shared resource. The only deadlock scenario possible with fewer entities is when one process tries to acquire a resource which it already owns, and that case can be avoided by the implementation of the synchronization primitives.

3.1.3 Livelock

Under livelock, in difference to deadlocks, a system is locked in an unintended loop of instructions that do not allow further computations on the intended task. Tasks that suffer from livelock still performs operations, but the operations have no other than administrative value and no real work is being performed. Note that the loop mentioned earlier does not have to be infinite, it may suffice with a finite number of iterations in order to severely degrade the performance of the system, or (in the case of real-time systems) even cause the system to fail (see also Section 3.3.2).

One example of livelock is given in the functionality of Ethernet (IEEE Std 802.3 [12]). Ethernet uses a Carrier Sense Multiple Access protocol with Collision Detection (CSMA/CD), it is in is behavior in case of collisions that we find a potential livelock situation. A collision can occur because the Carrier Sense part of the protocol cannot sense if two stations commence their transmissions at approximately the same time. As a collision occurs, all packets that where being transmitted at the time of the collision are destroyed, thus they will have to be re-sent. However, there is no mechanism in Ethernet that prevents that all or some nodes will be involved in a collision also the next time a package is sent, and the next and so forth. However improbable, this rock-paper-scissors[1] procedure give rise to a livelock, would it ever occur, and it must not go on for ever in order to present a serious bottleneck in the system. The probability of a livelock in these systems increase with the number of transmitting nodes in the system, and their rate on network packet production.

3.2 Race Conditions

The popular description of a race condition is as follows: when two or more system entities[2] potentially may be competing for resources at some time during execution, a race condition exists. These conditions may cause the system to behave very differently from time to time, depending on which entity that wins the race. This is of course very true, but also another type of race condition may occur.

An example of a race can be found in network communication, see Figure 3.2. Assuming that we have three nodes that are interconnected by some packet switched network which also serves several additional users that do not actively interact in this example, but still utilize the network resource. As two nodes of the three nodes, $Prod_1$ and $Prod_2$, produces one message each at approximately the same time, it is not possible off-line to determine which message that will be received first by the consumer $Cons_1$. Therefore, no assumption regarding the message ordering can be made in the consumer node in this case. The situation is normally referred to as a message race.

Netzer and Miller describes race conditions in [38], see also Netzer [34], where they search for race conditions in *prefixes* [34, s3.3.1;p21] of a particular execution. A prefix P' to an execution P has the same input as P, and the initial part of the ordered sequence of events in P' does not diverge from that of P in other aspect than that it may be a shorter sequence. After that initial sequence, the event histories may differ.

Prefixes are ordered into different sets, see Figure 3.3 which is reproduced from Netzer [34, Figure 3.1;p24], arrows are used to represent shared-data dependencies in the figure. We see that from the original execution seen in part (1.) in Figure 3.3, which exhibits the *actual race* (see below), we can also find

[1]The author, who is of Swedish origin, notes that this classic child's play is called "Sten Sax Påse" in Swedish.

[2]Entities can be such as tasks, threads, or processes.

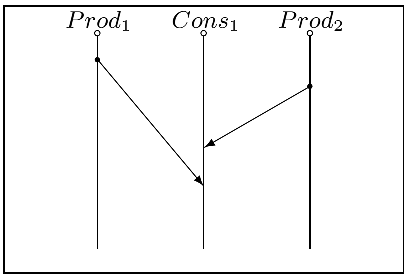

Figure 3.2: Example of a Message Race

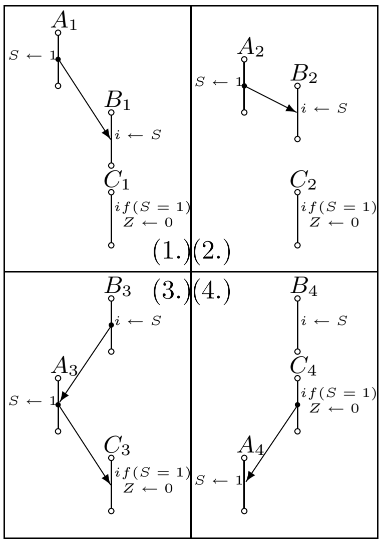

Figure 3.3: Data Races [34, Figure 3.1;p24]; (1.) Actual Execution. (2.) Feasible Execution. (3.) Feasible Execution. (4.) Infeasible Execution.

other execution orderings that are prefixes of the original execution. Part (2.) in Figure 3.3 shows an example of a feasible execution with the same event history as the actual execution, the execution is also a prefix of (1.). Also part (3.) is a feasible prefix of the original execution, but has an event history that diverges from the original execution. But part (4.) of the figure shows an unfeasible prefix, since the execution violates (implicit) dependencies in the system.

The authors identify two different classes of races: general, and data races. Where a general race is a situation where entities compete for resources (causing potentially unintended nondeterminism) in such a form that the ordering between two events is not guaranteed, there would then be a race between the two events. Note however, that many applications, or at least some parts of some applications, require the intentional use of general races. A data race is a violation to the atomicity of an operation on a shared resource, and is never intended.

Using the notion of prefixes, each race in a prefix of the original execution may then be *actual*, *apparent*, *feasible*. Races are classified according to the set-classification of the prefix, and their being general or data races.

Thus, a feasible data race is a data race that could really have happened to one of the feasible prefixes of the execution (in Figure 3.3 (1.)[3] or (2.)). Actual data races exists if and only if there exists at least one data race in the original execution (in Figure 3.3 (1.)), and it is not, in difference to feasible data races an NP-hard task to locate them. An apparent data race is a race that seems to be feasible, but implicit synchronization in the system prevents the occurrence of the race (in Figure 3.3 (4.)).

In equivalence, a apparent and feasible races can also be general. But there is not an equivalent to the apparent race in the general case as general races are experienced between program executions, and not within one execution.

Race conditions occur extremely frequently in for example shared memory systems, where they potentially occur at each unsynchronized access to shared variables by two or more different tasks. Considering that one must know the outcome of each race in order to recreate the system execution, the log of such races will grow quickly. Based on this observation, Ronsse et al. developed a method called RecPlay [42, 45] which uses the ROLT method described in Section 4.5.2. RecPlay can detect unwanted race conditions during the reproduction of the system execution, and may neglect to record vast amounts of information about potential races on-line. Confusingly, they use a differing terminology than that of Netzer and Miller which was described above.

Ronsse et al. differ between *Synchronization Races* which are intentional, and *Data Races* which are unintentional. This implies that some of the general races that Netzer and Miller defined, namely the unintended general races, are data races according to Ronsse et al.. In synchronization races, tasks race to gain access to shared resources, where as data races occur when synchronization is insufficient. It is data races that should be located and removed. By reproducing the execution several times, an identified data race is pinpointed, and sufficient information is gathered to explicitly identify its source. Of the three execution reproductions made, the first pass senses the presence of a data race. Thereafter, a second pass identifies the data address where the data race occurred. Finally a third pass can identify the issued instructions that cause the data race by operating on the memory address.

Focusing on data races, the RecPlay fails to direct other sources of errors in parallel systems, the method does not direct how to facilitate the replay of a system when the initial state is lost or there are gaps in the monitoring history. It is therefore not feasible to use the method in systems where memory resources are small relative to the required up-time of the system.

[3] In the case of part (1.) in Figure 3.3, note that an execution can have the same sequence of events - without being equivalent to the original execution in all other aspects.

3.3 Real-Time Errors

In this section we shall review problems that normally arise only in real-time systems.

3.3.1 Violations to the Order of Precedence

Precedence orders are relations that constitute dependencies between events. These can also exist in non-real-time contexts, but as they are a natural ingredient in practically all real-time systems, they are reviewed in this section.

The orders are typically on the form "Event A must occur before event B", where events often are task executions. These precedence orders can be quite complex and consist of many different events, they are often referred to as *precedence graphs.* Note that one system may have many independent precedence graphs.

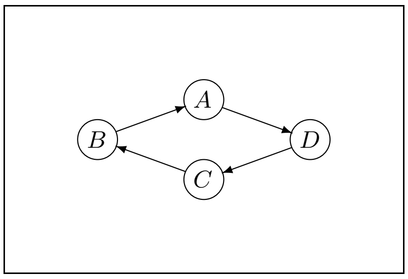

Figure 3.4: A Precedence Graph

As an example of a precedence graph we turn to the manipulation of external devices. Device D is a part of a real-time system which also contains the tasks A, B, and C. The device is controlled by task A, which receive orders from task B which makes decisions based on information about the device state sampled by task C. After that the device has received a command from C via A it will take until time t_d before the command is completed.

Because of the inertia in the system, it is important that the control decisions from B are not issued too frequently. A registered deviation from the expected result cannot be certified until t_d time units after that the last control command was issued to the device by A.

Thus, there exist a precedence order between the actions taken by the tasks in the system. No control command may be issued before a valid sensor reading has been acquired from C. Thereafter, samples are invalid until the sensor readings have propagated to B, B has taken appropriate action in the form of a command, A has transferred that command, and D has reacted to it. The precedence graph for the system is displayed in Figure 3.4.

3.3.2 Timing Errors

In the context of real-time systems, it is not only required that the functional aspect of the program is correct, but also that the timing of the system follows certain rules defined by the system specification. A real-time system has certain timing constraints, which can be more or less complex. Timing errors may be caused by other errors, described above, for example a livelock or a deadlock can force a system to violate its temporal requirements. But there are also other, more intricate causes that will be described in this section.

Tsai et al. provides a classification of causes of timing errors in [62, s9.1.1;p192]:

Computation Causes If a greedy task requires more resources than has been granted, other tasks may find them selves with to little resources to compete their task. This problem can easily arise, should the measured Worst Case Execution Time (WCET) be lower than the real WCET. Best results are achieved by estimating a WCET which is as tight as possible, but never too optimistic. As the name implies, measured WCET is determined by measurement, a process which may have poor coverage, this is a very likely cause of errors. An estimated WCET can also be calculated, a method that is compromised by the use of multitasking programming, caches, pipelines, and/or superscalars. Also errors in sequential programs may cause this type of error, see for example control errors at the beginning of Section 3.

Scheduling Causes Related to the above cause, errors in the scheduling of the system may also cause the system to validate its timing. This problem could arise if the schedulability analysis has not considered all possible parameters. If it is estimated, say, that an interrupt will occur at most once each 50 milliseconds. If there, in reality, is 45 milliseconds between each instance of the interrupt, the system may prove to be un-schedulable. Note that the WCET of the interrupt, and all other parts of the system, may still be correct. Also other scheduling related sources of errors exists, such as the occurrence of jitter in combination with end-to-end deadline constraints.

Synchronization Causes The occurrence of synchronization problems have been covered in previous sections (see for example deadlocks and livelocks in Section 3.1), and they too may of course cause a real-time system to violate its temporal restrictions.

Thus, timing errors arise as a consequence of previous errors, some of which are only considered as errors in real-time systems.

Chapter 4

Monitoring Execution Traces

Monitoring, according to McDowell and Helmbold [30], is the process of gathering information about a program's execution. By monitoring the execution of a program we can analyze that execution off-line in some form of model of the platform that was used, an issue which will be covered in Section 5 - the current chapter will deal with the problem of performing monitoring. Normally, monitoring is performed either by additional software that is added to the system at some level, by tailored hardware, or by a hybrid approach. Each approach has its advantages and drawbacks.

Because monitoring provides us with detailed information about a systems execution, detailed enough to recreate the execution, we can apply cyclic debugging to a monitored system. By recording significant events, whose occurrences cannot be definitely determined offline, we may alleviate all the problems of cyclical debugging that where presented in Section 2.6.1.

4.1 The Probe Effect and the Observability Problem

There are similarities between the probe effect and the observability (or observer) problem. In this section we explain the two, and point out differences and similarities. We shall also discuss a problem that previously has shared name with the observability problem, as it has no other relation to it, we shall rename it in order to avoid confusion.

4.1.1 The Probe Effect

The *probe effect* [9], which is another name for *Heisenbergs uncertainty principle*[1] when applied to software engineering [24, 30, 50], can become visible when code is added or removed to a system, or the system is modified in some other way that will imply increased execution times. Modifying the system in any way may alter the timing in the system. Extra code will require computing- and other resources, the removal of code will free resources that can be used by tasks that would have been blocked, and modifications to data may change the program flow. Differences in the temporal behavior may in turn result in that the modifications have a different result on the system performance than expected.

It is quite convenient to use real-time systems when exemplifying the probe effect. Imagine a system of two tasks that compete for execution resources, where some synchronization problem exists between the two tasks. Say that the two tasks control an external process, but that one of the tasks occasionally issues control commands too soon after that the previous task has issued a command, thus preventing the previous command from effecting the external process as intended.

In order to debug the system, we would like to probe into the internals of the tasks so that we could determine the cause of the problem. However, if we perform this probe by inserting some auxiliary code (code that does not aid the progress of the system) that will monitor the system, that code will effect the system. If we are unlucky, it will do so in a way that the time between the two control commands is lengthened, thus causing the bug to disappear during some executions which may very well be just that subset which we examine. If we then remove the probes, the bug may reappear. Also the opposite is possible, by adding probes to a system, we may cause errors to appear that where not previously present. Also a combination of the two is possible, by adding probes to the system, we may remove one error, only to invoke another.

The last example is perhaps the most intriguing, we may then find ourselves identifying the wrong bug, and correcting that one instead of the real one. This problem should be detected by a regression testing procedure (see Section 5.3).

Debugging is not the only situation in which the probe effect may effect the system, it is also possible that modifications to old systems, or bugfixes, cause the same problems. One may view it as that the removal of code is equivalent with removing a probe from the system, and that adding functionality can cause the same problems as adding a probe to the system. A general rule is that if the source code is modified, probe effect related problems may arise.

There are however two exceptions to this rule.

Schütz notes in "Fundamental Issues in Testing Distributed Real-Time Systems" [48] that it is possible to remove code if the only consequence of the removal is that the idle task of the system will receive more execution time. However, this is rather hard to ensure unless the system is time-triggered. Schütz states that,

[1]They have also been called Heisenbugs [45].

in a time-triggered system, provided that the scheduled execution slot of the task that is to be removed is not adjacent to the slot of any other task (except the idle task), the task is in a *temporal firewall*, and may be removed without consequence to the remaining system. This is provided of course that the task does not perform any work that is used by other entities in the system.

The second exception has been noted by Thane in his PhD dissertation [56, s4.3.3;p42]. Thane stated that code can be removed if the only consequence of the removal is that the idle task of the operating system receives a larger percentage of the total system execution time. In order to satisfy this requirement, the task from where the code is removed must have the lowest of priorities among the (other than the idle task), and it must be established that the task never blocks the execution of other tasks remaining in the system. Thus, the task from where the probes is removed cannot control mutual exclusion or communication primitives, such as semaphores or other, shared with tasks remaining in the system. The use of schemes such as direct inheritance or similar for deadlock avoidance will limit the use of such primitives even further.

4.1.2 The Observability Problem

The observer problem, described by Fidge in "Fundamentals of Distributed System Observation" published 1996 [8], describes the problem of obtaining a truthful view of the events in an observed system. For example, as a distributed system is being observed, if the observer cannot be tightly coupled with the system it is observing, problems related to the observers apprehension of the ordering of events on different nodes may occur. Depending on variations in the propagation time of observer notifications, the ordering of events may be confused.

According to Fidge, we may divide the observer problem into at least four sub-problems [8]: (1.) multiple observers may see different event orderings, (2.) observers may see incorrect orderings of events, (3.) different executions may yield different event orderings, and (4.) events may have arbitrary event orderings. All are more or less results of the absence of an exact global time-base, and/or the fact that network propagation times are not constant. Because of the lack of a exact global time, we cannot rely on any time-stamp taken at the node where the event occurred, if the observer is situated on another node.

1. In a system where many observers are used, different observers may see different event orderings, because they propagation of the event notification requires different time to different destinations.
2. As the propagation through a network may differ between two network packages, a package that is sent after another may arrive earlier. Thus, if two events occur on different nodes at different times, the notification of the last event may arrive at the observer before the first notification has arrived, thus erroneously implying that the last event occurred before the first.

3. Because the clock rate of each node will diverge slightly from the ideal clock and the other clocks in the system, and the rate of that deviation partly depends on environmental aspects, even different invocations of a distributed system will differ.

4. Some of the events in the system are unrelated, and may therefore be allowed to occur in arbitrary orderings. The problem with this is that an observer must know and recognize that, as different tests are run, it is allowed to have differing orderings between some of the events.

Item number (4.) in the list above is related to Polednas PhD dissertation "Replica Determinism in Fault-Tolerant Real-Time Systems" from 1994 [41]. Poledna direct the problem of *replica determinism* when using redundancy as a mean to increase the fault-tolerance of a real-time system. In other words, he directs the problem of ensuring that two components that are supposed to perform the same task have the same behavior when they are operating correctly. This is related as (4.) describe that we must be able to correlate executions that are temporally differentiated, and Poledna does the same for spatially differentiated executions.

4.1.3 The Probe-ability Problem

It should be noted that Schütz also discuss a subject which he calls observability [48], but which has a slightly different definition which is closer related to the probe effect described above. Schütz states that a system must be *observable*, meaning that it must be possible to extract sufficient information from the system. Another, equally suitable, term is "probe-able". What is "sufficient" is determined by the present fault hypothesis. In the remainder of this report, mentioning observability implies Fidges observability described in Section 4.1.2, and the problem described by Schütz shall be referred to the probe-ability problem.

4.1.4 Conclusion

Thus we may conclude that the probe effect causes changes to the program execution, whereas the observability problem affects our perceived view of the program execution, and the probe-ability problem directs the problem of being able to observe. The first and second of these are however related in that it may be difficult to differentiate between problems resulting from probe effects and problems resulting from the observability problem.

4.2 Measuring Consumed Computation Resources

If the logging of system events is to be used in debugging purposes, it is important to relate events to software execution. It must be possible to state how mush execution resources a task has consumed between two entries in the log. There are at least two ways of doing this, one is to use a hardware platform which supports instruction counting, cycle counting or similar, the other is to use a software implementation.

An example of a hardware solution is implemented in the Intel x86 architecture. A processor cycle counter is accessible through the use of the assembler instruction `RDTSC`. Note however that this implementation is not reliable in architectures such as Pentium II, Pentium Pro, and onwards. The reason therefore is that more advanced models in the x86 family use out-or-order execution which can lead to pessimistic or optimistic measurements.

In their article "Debugging Parallel Programs with Instant Replay" published in 1989 [31], Mellor-Crummey and LeBlanc present a method that can instrument assembler-code with counters, thus enabling the counting of executed instructions, the method is called Software Instruction Counter (SIC). The authors note that the code of a program consists of short sequences of sequential code, called basic blocks, and conditional, or unconditional, connections between some of the basic blocks (by branches, jumps, or function calls). These one-way connections can either connect a basic block with a later (with higher address-value than the present), a forward branch, or with a prior block, a backward branch. To uniquely mark each instruction instance that is executed, the authors state that a combination of the program counter value and the number of backward branches is sufficient. They can therefore construct a low-cost software-based instruction counter which only resource requirements except a small computation overhead is a reserved data-register which is used solely for performance reasons.

4.2.1 Consistent Temporal View

An issue arising when trying to relate several executions on different nodes is the lack of a synchronized global clock [30]. As events occur on concurrent nodes, some system architectures cannot produce a correct order between them. Tightly coupled parallel systems, and multitasking single-node systems, are able to do so, because all system entities depend on the same real-time clock [62, s3.1;p51]. But, because of the observability problem (See Section 4.1.2) distributed systems can only make weak assumptions about the ordering of events provided that they do not use an algorithm for global clock synchronization [17].

Ordering of events can be either *partial*, or *total* [62, s2.1;p30]. Where partial order describes the local sequence of events (in our context locally is on a specific node), and total order describes the global order of events. Thus, unsynchronized systems cannot determine the exact total order of events, but they may be able to find an estimation of the global order by using a method for clock synchronization, or logic clocks [19]. For any reasonable failure semantics, a total order of events must be described if a distributed system is to be debugged. If such an order cannot be established, the overall understanding of the complex inter-node-relations is lost, wherefore the system can only be debugged node per node.

In the classic paper "Time, Clocks, and the Ordering of Events in a Distributed System" by Lamport in 1978 [19], the author describes a now classic method for implementing a logic time-base in systems that lack a global time-base. The method, normally referred to as *Lamport clocks*, is based upon the counting of events, its purpose is to derive a total order on all events in the system (where the definition of an event is application specific). Each node and each shared object that implements the method has its current opinion of the time stored. As a significant event occurs, it is given the time-stamp equal to the largest of the current local clock value of the node and the shared object, plus a value which normally is one (1). After which the local clock value of both the node and of the shared object are set to the same value as the time-stamp.

Another classic paper that directs the problem of synchronization in distributed environments is "Clock Synchronization in Distributed Real-Time Systems" which was written in 1987 by Kopetz and Ochsenreiter [17]. The paper presents an algorithm for global clock synchronization.

4.3 Global State

One big problem that one has to face when implementing a strategy that uses monitoring of a system is that the initial state of the system must be known in order to understand the context of the events recorded by the monitor. In some real-time systems, this can easily be done by using the Least Common Multiple (LCM) of the periods of the tasks that reside on a specific node. That LCM would describe the periodicity of the system, and in some systems, these LCM's can be said to be individually unrelated.

For example in a simple control application, it may be possible to view one sampling-actuating loop iteration without knowledge of outputs and flows in all prior iterations. Note that many systems are not this simple; it is common that there exist some relation between iterations wherefore the scheme cannot be used without adaptation. Such an adaptation may be checkpointing of some global variables at the end of the execution of an iteration.

In other systems, which are not periodic in the execution characteristic, obtaining a known state may imply that the entire system must be incorporated into a giant consistent checkpoint.

In the prior case, assuming a checkpoint has to be made, the size of the checkpoint is expected to be smaller then the latter. We build this assumption on the thesis that there exist a relation between the size of the checkpoint and the implicit knowledge of the system activities.

However, assuming that checkpoints are used, only making one checkpoint in the beginning of the simulation is not sufficient. Because very long monitoring sessions require very much memory resources in order to keep the logs, and those resources are finite, it is required that old log entries be evicted as the memory is exhausted [53]. The eviction is made in favor for newer entries, that have a larger relevance for the current propagation of the execution. In other words, a circular queue ADT (abstract data type) could be used for logging the messages. Thus, we cannot assume that we will always be able to start simulating the system from the beginning. In fact, we may not even desire to do so; as it may take a very long session to produce a fault that we wish to examine, and simulation is much more demanding than native execution [10, s4.4.4;p58], it may be profitable to be able to start the simulation in the middle of a trace. Netzer et al. has directed this problem in their Incremental Replay approach (see Section 4.5.2).

Note however, that there may be better solutions than a simple circular queue. Messages could be assumed to have a timespan in which they are important for the system execution. At the end of that timespan, they can be evicted without consequence for the replay. It is not necessarily so, that the lengths of that timespan is the same for all types of entries, wherefore other structures could be preferable (see Section 6.5).

4.3.1 Checkpointing

The reason for making checkpoints of a system is to be able to start over with the execution at some later point [37, 65]. There are to our knowledge three main applications for this ability: The first case applies to systems that can sense an error in their execution, and as a response to this can decide to roll-back and try again. The second case applies to systems that have some source of non-determinism in them; in order to apply cyclic debugging strategies to these systems, a monitor - replay approach can be used. The third, and final, application is to allow deterministic testing of non-deterministic systems. Sources of non-determinism may be race conditions due to some level of parallelism, or other. We can differentiate between applications that need to recreate a system state in that the first performs on-line, where as the second and third are applied off-line. Also, on-line recreations must not necessarily receive the same inputs as the execution that was recreated, whereas the sole purpose of the second and third application is to recreate the system with as much adherence to the original execution as possible.

Zambonelli and Netzer [65] state that the use of checkpointing is always required when recreating a system state. We argue that this is at least dependent on the task model used. Considering for example a terminating task model similar to that implemented in the Asterix real-time operating system presented by Thane et al. in the article "The Asterix Real-Time Kernel" published in 2001 [58]. As a task conforming to that model is always terminated at the end of each instance (the alternative is usually to issue a relative sleep-command), there is no need to save its state. Only the input parameters to the next instance are required, but so are the input states to new tasks in a non-terminating task model.

Recovery Line

For which ever reason, restarting the execution of a system is only feasible if certain requirements on the point from where the system is started are fulfilled. Chow and Johnson formulates in [4, s13.1;p510] the requirements for starting points used in replay or recovery of distributed systems:

> "The restarting state of any processor should not casually follow the restarting state of another processor."

The quote captures, in one sentence, the requirement that the starting point must be a fully consistent state in the execution of the system. All messages, and other events, that are in transit (i.e. sent but not received) must be known, and there must be no messages that are received but not sent if they cannot be deterministically recreated. The latter of the two, messages that are received but not sent, are normally referred to as *orphan* messages.

Another, equally beautiful phrasing of this condition was formulated by Wang and Fuchs in "Optimal Message Log Reclamation for Uncoordinated Checkpointing" [63]:

> "... we define a *consistent global checkpoint* as a set of N checkpoints, one from each process, no two of which are related through the *happened-before* relation."

The happened-before relation mentioned in the quote was defined by Lamport in 1978 [19].

The states, or set of distributed states, that fulfill the constraints that are placed on a feasible starting point for replay or recovery is normally referred to as a *recovery line* [4, s13.1;p510].

Approaches to Distributed Checkpointing

The nature of distributed systems makes it hard to ensure that a recovery line can be identified in the logs of checkpoints, mechanisms must be applied that can alleviate the problem. According to Wang and Fuchs, there are mainly three different strategies to distributed checkpointing [63]: Uncoordinated checkpointing, coordinated checkpointing, and log-based techniques. Chow and Johnson divide the log based techniques into thee sub-categories: Synchronous logging, asynchronous logging, and adaptive logging.

Uncoordinated Checkpointing As there is no coordination between nodes concerning the timing of checkpoint acquisition, there are no guarantees for the existence of a valid recovery line. When trying to obtain a recovery line by selecting a set of checkpoints, one from each system entity (processor, process, or other), there is a (substantial) risk that a pair of checkpoints in the set are inconsistent. There are two different scenarios; One scenario is that the checkpoint at the receiving entity represent a state when a particular message cannot not yet have been received, but the checkpoint at the sending entity represent a state when the message must have been sent - i.e. the message is in transit. The other scenario is that the checkpoint at the receiving entity represents the state when a message must have been received, but the checkpoint at the sending entity represent a state where the message cannot have been sent - the message is referred to as an *orphan* message.

As such a set of checkpoints validate the requirements for a recovery line, other checkpoints must be chosen, there are however no guarantees for that the next set of checkpoints are consistent, and so forth. This undesired effect is referred to as the *domino* effect or *cascading rollbacks*.

Coordinated Checkpointing The main contribution of coordinated checkpointing is that there each acquired checkpoint is a member of at least one recovery line, thus alleviating the problem of cascading rollbacks.

Synchronous Logging Logging messages that are sent in the system is also a form of checkpointing. In synchronous logging, each message is logged before it is delivered. This can be said to be the easy way out, there are other more troublesome logging policies.

Asynchronous Logging In difference from synchronous logging, asynchronous logging allows the activities of logging messages and delivering them to execute in parallel or out of order. Problems will arise due to this more relaxed policy, but the advantage lies in lower latencies in package delivery.

One problem with asynchronous logging is of course that all messages are not always in the log after a system halt or crash. If the system stops, or experience a severe failure in the logging mechanism, as a log-message is in transit, the log does not reflect the complete system execution. Threatening to prevent system replay, this situation can be detected using *dependency tracking* [54], that is to track the dependencies between checkpoints on different entities.

Adaptive Logging It is not always required to log every single message in order to recreate a system state. Adaptive logging mechanisms can identify which messages can be ignored.

As we can see in this description, some approaches optimistically hope that a recovery line can be found in the available data collected, and some others pessimistically ensure during run-time that such a line will be found. The advantages of the latter class of approaches is that it is ensured that a replay is possible, but the drawback is in run-time performance. For the first class, optimistic approaches, the opposite is true.

4.3.2 Control- and Data Flow

Platter is to our knowledge the first to differentiate between system entities when discussing monitoring of computer systems. In the article "Real-Time Execution Monitoring" [40] from 1984, he defined a process state to consist of two parts: the *data-* and the *control substrate*. The data substrate represents the data structures currently under control of the process, while the control substrate represents the current point of execution.

Thane [56, s4.2;p37] classifies monitoring subjects into three categories: Data flow, Control flow, and Resources. Where the data flow concerns the flow of data between different architectural components on some level. The control flow is an abstraction of the path taken through a system - this could for example be described by the ordering and timing of events and interrupts, the results and timing of task switches, and other issues that can describe the execution flow. The last category, resources, describes the uses of shared physical resources. We can log CPU utilization, memory use and other issues.

The control flow of the system consists the sequences of instructions executed by the processors(s), and relevant[2] timing information regarding that execution. The data flow of the system is represented both by the relevant[3] alterations of system data during time, and timing information regarding these alterations. In order to successfully replay the monitored system, both the control- and the data-flow must be covered during monitoring.

4.4 Scope of Monitoring

The scope of a monitoring activity must be well-defined, if the scope alters, this will give rise to a probe effect. This effect may or may not be visible, but to this day the only general[4] way to guarantee that the effect of altered monitoring scope is negligible is by using exhaustive testing.

Implied by the scope of the monitoring activities, and the prior knowledge of the system, is the level to which the system execution is known, and therefore also which types of errors that can be located, analyzed, and corrected. If the monitoring is exhaustive all thinkable errors can be debugged, but every abstraction opens the door for errors to escape the debugging process unnoticed. Thus, we must have a fault hypothesis (see Section 2.3) before we can define the monitoring activities in the system.

[2]What timing information that is "relevant" here is defined by system interactions. Timing is only relevant if two subsystems affecting each other, through communication or other interference.

[3]What data operations that are "relevant" is defined by what cannot be reconstructed by deterministic re-execution of the software.

[4]Remember the temporal firewall presented by Schütz [48] which allows guarantees in a very special case.

4.4.1 Logging

The product of a monitoring activity can be logged to a consistent data storage, thus creating a log of an execution. The contents of the log at a given time, together with a knowledge of the system and a system model, can allow us to replay the monitored execution of the system.

An important factor that will influence the design of a system is the amount of memory resources required to keep the log.

We have previously (in Section 2.2) defined the terms fault, error, and failure. In order to debug a system we must be able to follow the propagation of an error to a failure. The time from the execution of the error until it has propagated to a failure is the *incubation time* of a failure. The incubation time of a system, together with other factors, implies how long the log of the monitoring activities must be. Of course, the length of the log is important in finding our the memory resources required for the log. As the fault hypothesis defines which failures that may occur, it is an important factor when finding the incubation times of the system.

A factor which was consistently ignored in the above argumentation is the system knowledge required. This is a very important factor when defining the fault hypothesis, the incubation time, the length of the log, and the memory resources required to keep the log. It is therefore a pity that it differs so much between systems.

4.5 Discussing Monitoring Approaches

In this section, we will discuss and compare three different basic approaches to monitoring, software, hardware, and hybrid monitoring.

It is also possible to classify monitoring approaches based on how they effect the system during use, Schütz [48] states three classes based on how they handle the probe effect: by ignoring the effect, by minimizing the impact on the system during debugging, or by avoiding the probe effect. Classification into these three classes require inspection of particular implementations.

4.5.1 Hardware Monitoring

Hardware monitoring mechanisms are tailored devices, they need to be adopted to the target system, which suggests that this is a rather expensive approach. On the other hand, they do not have to intrude at all on the device functionality [62, s2.3;p37].

Basic approaches to hardware monitoring include bus snooping, to spy or listen to the messages sent over the system bus. The quantities of messages, and their relative size, result in that large quantities of data must be stored. Another problem with hardware monitors is that they must look at very low level information [62, s2.3.2;p37], the data that is visible has low information content relative to the program execution. That is to say that a single bus message can not say much about the execution of a program, whereas (for example) the name of the current state can say a lot about the traversing of a state-machine. It is then up to off-line methods to interpret the collected information that is output from the monitoring process, correlate them to the system software and hardware, and translate the result into a format that is understandable to humans [16]. Needless to say, the amount of information may be quite extensive, but this problem is more or less inherent in the monitoring methodology. Also, implementations, and to some extent even solutions, are platform specific. Furthermore, advances in hardware technology makes it more and more interesting to integrate solutions to a single chip, so called System-on-Chip (SoC) solutions [62, s5;p103]. SoC solutions limit the insight to the internals of the system, and it is therefore more difficult to construct hardware monitors for these systems provided that they are not incorporated on the chip [16]. A solution could be to move also the monitoring into the chip, but this is approach is of course only available to the designers of the device. Thus, SoC technology is obstructing the use of off-the-shelf components where monitoring is required. We shall, in Section 4.5.3, survey a proposed methodology for SoC monitoring.

Boundary Scan IEEE Standard 1149.1 defines test logic [13]. The standard is a result from work by the Joint Test Action Group (JTAG)[5] The Boundary Scan method can be used to test Integrated Circuits (IC's), interconnections between different assembled IC's, and to observe and modify the operation of an IC. Provided that the processors of the system implements Boundary Scan, it is feasible to force reproduction of a execution through the use of that interface. The reproduction method could provide the data and instruction flow through the Boundary Scan interface, and force execution of the correct instructions with the correct data. On the positive side, this allows us to have a reproduction facility on the real hardware, without modifications to that hardware. However, the Boundary Scan interface, through which all data and all instructions is to be feed, is a serial interface. Also, as the pins of the circuit, which Boundary Scan can control, are connected via Boundary Scan as a large shift register, causing the propagation of the signals to be very slow. Thereby inferring large temporal penalties on the reproduction of the execution. In the case of monitoring, it seems that the same problem provides a limit for the granularity of the monitoring process, the serial interface constitutes a severe bottleneck.

In their article "Emerging On-Chip Debugging Techniques for Real-Time Embedded Systems" published in 2000 [29], MacNamee and Heffernan discuss the issue of On-Chip Debugging (OnCD) with a state of the practice point of view. OnCD has the capability of addressing the problem of monitoring complex processor architectures, especially those with on-chip caches, as it uses monitoring hardware that reside inside the components. However, solutions available today lack real-time capabilities in for example memory monitoring (an example is the Motorola ColdFire). The lack of real-time monitoring of

[5]The group has a homepage at www.jtag.com.

memory resources can be explained by the fact that real-time monitoring requires the monitor to be prioritized over the application, thus leading to intrusive monitoring.

Logic Analysers are often used to monitor the behavior of hardware components. There are many devices available on the market, they have the capability to hook on to, and monitor, buses that transport data or instructions between physical modules of a system. On the positive side, logic analyzers are not intrusive on the target functionality, not event in the temporal domain. However, traces available are very low-level, and not all required information may be available. Systems that have very integrated designs, perhaps with on-chip caches, or even multiple processors on one chip, do not pass all required information on buses that are physically available for the logic analyzer [16]. But the fact still remains that logic analyzers are used in many commercial projects, and even though they cannot solve all problems, or even provide good solutions to all of the problems that they can solve, they among the better solutions commercially available for debugging real-time systems today.

Several of Motorola's (www.mot-sps.com) MicroController Units (MCU's) support the Background Debug Mode (BDM) [11] interface, this interface is utilized in their EValuation Board (EVB) products that facilitate remote debugging of the MCU's. The BDM interface allows an user to control a remote target MCU and access both memory and I/O devices via a serial interface. BDM uses a small amount of on-chip support logic, some additional microcode in the CPU module, and a dedicated serial port.

The BDM interface is constructed of different instructions which can be issued in order to examine the state of the device. Instructions may be either hardware instructions, in which case they are not necessarily very intrusive on the functionality of the device, or they may be firmware instructions, which are intrusive. Hardware instructions allow reading or writing to all memory locations of the device, these operations are initially given the lowest priority, i.e. they are only executed if no other instructions are pending, but a fairness policy is used if the instructions are not issued within a predefined time. Firmware instructions must be issued in a special firmware-mode, and then the debugger can read and write registers on the device.

Motorola also provides a On-Chip Emulation (OnCE) interface with some models, the interface combines features of BDM and JTAG debugging.

Domain Technologies Inc. (www.domaintec.com) provides a tool called BoxView that is based on the Boundary Scan and OnCE technologies. Several BoxView devices can be connected via a BoxServer so that multiple targets can be controlled synchronously. If OnCE is used as a method of debugging, systems of up to two nodes can be debugged. In JTAG mode that number is 255. Note that this approach does not use a reproduction approach to debugging, and therefore is not suitable for real-time systems. Agilent Technologies (www.agilent.com) provides a large range of logic analyzers and processor specific high-level language debuggers, but they do not use the reproduction approach either. They do however, allow non-intrusive data and control flow monitoring with the possibility to correlate spatially differing observations to the temporal domain.

The Nexus 5001 standard (www.ieee-isto.org/Nexus5001) [29, 52] describes a hardware solution that supports debugging and tracing of embedded systems, it also supports debugging of superscalar and pipelined architectures. We will in this section provide information on selected parts of the standard.

There are four different classes of compliance in the Nexus 5001 standard (1 - 4 where 4 is the strongest), class 2 must have a Boundary Scan interface, and class 2 - 4 must have a standard specific connection called AUX (however, they may also optionally implement a Boundary Scan interface).

The AUX interface is a parallel medium with 1 - 16 pins, the bandwidth requirement of the implementation may dictate the width of the AUX interface. It is a packet based medium, which result in that packet-arrival-times cannot be determined at the time of transmission. Therefore, assumptions may not be made of the relative order of, for example, a change of ownership and a taken branch.

There are three different tracing mechanisms available in the standard:

Ownership trace Implementations of class 2, 3, and 4, must support ownership traces which can monitor process ownership while the processor runs in real-time. This provides a macroscopic view (of task orderings etc.), can be used to monitor ownerships of shared resources such as code pages in a virtual memory system etc.

Program trace Class 2, 3, and 4 type devices must provide a facility that allows monitoring of program flow while the processor runs in real-time. A completely hardware controlled operation, the information is flushed via the AUX. At the occurrences of branches and exception (also known as program flow discontinuities), trace information is passed to the system observer via the AUX medium.

Program trace messages can be of two types, either direct branch messages, or indirect branches which can also concern the occurrence of an exception. The difference between the two is that direct branch messages are self-contained, and indirect messages are related to the previous message that was sent. Using long sequences of indirect messages in long traces can result in that the loss of information (as a consequence of space exhaustion) reduces the ability to reconstruct the execution. To alleviate this problem, certain events can be set to trigger the use of direct messages, something which is also triggered periodically at the minimum rate of every 256 program trace message.

Data trace To monitor memory operations while the processor runs in real-time, class 3 and 4 implementations must provide the possibility of tracing writes, and may optionally trace also read instructions.

The standard also specifies that devices of class 3 and 4 must allow read and write access by the debugger to any memory location during run-time as well as when the execution is halted. It is up to the implementer to determine through which interface this facility is accessible.

4.5.2 Software Monitoring

Similarly to the cyclic debugging approach described above, software implemented monitoring is also vulnerable to the probe effect. That probe effect may, however, be avoided by allowing traces to remain inside the release version of the program [60] [62, s3.1;p51].

Remaining probes will of course cause performance degradation, but one may argue that they shall remain also because this allows us to introduce a form of black-box to the software, similar to that of airplanes. The black-box may then be used if a released program experience a failure during execution. However, Kranzlmüller [18, s4.2.1;p84], pointed out that the monitoring activities need to be defined quite early in the design process, and that the managing of the monitoring data may present a problem.

Software monitoring can either be performed at system, or process (task) level [62, s3.5;p68]. Monitoring at system level enables the monitor to see operating system specifics in the system. It is possible to view many of the data structures that effect system performance, such as Translate Look-aside Buffer (TLB) entries that describe the mappings between virtual and physical memory, also task control blocks, semaphore queues, and many other data structures are visible. Issues related to the control flow of the system that are visible on system level include interrupt occurrences, task switches and paths through code within system-calls. Monitoring at the task level will not allow monitoring of these, but other possibilities are open, such as events related to the specific task that is monitored. Concerning the data flow, we can observe local and global variables, and of the control flow, we can record the executions flow through a program.

Thane [56, s4.3.3;p41] describes four architectural solutions for software monitoring: *kernel probes*, *software-probes*, *probe-tasks*, and *probe-nodes*. Where kernel probes can monitor operating system events such as task-switches and interference due to interrupt occurrences. Software-probes are additions to the monitored task, they are auxiliary outputs from that task. Probe-tasks have as their only functional objective to monitor other tasks, either by cooperation from software-probes, or by snooping shared resources. Finally, probe-nodes are dedicated nodes that either snoop the communication medium used by other tasks, or receive input from either software-probes or probe-tasks.

Stewart and Gentleman [53] recommend the use of data structure audits, a construct which is also described by Leveson in [25, s16.4.1;p419] where it is also referred to as independent monitoring. An auditor could for example check whether a data structure is self-consistent, or simply logging its changes. Auditing can be performed by a probe-task, also known as a spy task, and can be a more or less complex operation.

Instant Replay was presented in 1987 by LeBlanc and Mellor-Crummey [23]. The method aims at facilitating replay for tightly coupled systems, but it is claimed to be extendible also to loosely coupled systems. They make no assumption about the availability of synchronized clocks, or globally-consistent logical time. By providing the same inputs to the system, and logging the relative order of accesses to shared objects, the repeatibility of the system is ensured. However, as Instant Replay performs best if it can be assumed that there are available high-level communication primitives that can be assumed to be correct. In other cases, each individual memory reference must be logged, thus leading to large logs. As the method monitors the accesses to the shared objects on a very coarse-grained level, they cannot detect data races inside these access sequences [35].

Logging algorithms in message passing systems must choose one of two main approaches, they can either log messages that are sent, or include all nodes that are transmitting messages in the reproduction of the system execution. Zambonelli and Netzer et al. discuss the situation in "Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs" and "An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications" [37, 65]. The authors state that logging all messages is resource demanding during the reference execution, but recreating all messages during the reproduction can be very demanding during that process. This is therefore a trade-off situation. They discuss whether it would be possible to make a compromise: If all nodes record sufficient information about their execution, save all external messages, to facilitate reproduction it is theoretically possible to recreate all messages that occur in the system by reproducing the execution of all nodes. Now, if it is judged that it would require large computations in order to recreate a particular message the message is logged, otherwise it is not, and must be recreated during the reproduction. The incremental replay approach also allows a replay session to start at a point which is not the starting point of the system. A feature which is very useful when the reference execution was long. Later, also Thane and Hansson [60] has provided this feature (see Section 5.4.3).

Netzer presented a method based on the Instant Replay method in two articles published in 1993, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs" and "Trace Size vs Parallelism in Trace-and-Replay Debugging of Shared-Memory Programs" [35, 36]. The objective of Netzers work was to improve the possibility of detecting races, and still minimize the logging of system events. The author argues that, as the computing capacity increase with respect to storage access time, it is favorable to trade log size to computation complexity. Viewing the interactions on shared objects as a graph, where accesses are nodes, and the flow is represented as edges, we can see that some of the edges are implied by the program flow. By transitive reduction of the graph, omitting all edges that are implied by program flow, Netzer is able to reduce the information required to describe the execution of the system. Ronsse et al. surveyed the approach in the article "Execution Replay and Debugging" [44], where they presented the following relevant disadvantages of the method: The use of vector clocks [1] limits the possibilities for dynamic task creation as the size of the clocks varies with the number of processes in the system. The overhead due to clock comparisons can be expected to be big.

Levrouw et al. presented the Reconstruction Of Lamport Timestamps (ROLT) method in “A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks” published in 1994 [26], an improvement of Netzers method described above. Instead of using vector clocks, as Netzer, the authors use Lamport clocks (see Section 4.2.1). The gain of using Lamport clocks lies in ease of maintenance, but it also opens a possibility to optimize the Netzer algorithm. Looking at the Lamport algorithm, there are two possible actions at the receipt of an event: Either the clock value of the local task is incremented by one, or it is replaced with the value of the shared object incremented by one. The former is a deterministic action, where as the latter in nondeterministic. It is sufficient to store a log entry only in the nondeterministic case. The penalty inferred by the use of this optimization is that a log entry must consist of both the clock value before the occurrence of the event, and the clock value after the event. During replay, the omitted logs can then be deterministically recreated. Ronsse and Zwaenepoel presents an implementation of the ROLT method on a Treadmarks [14] platform in [46]. The Treadmarks is a distributed shared-memory system.

DEEP by Veridan Systems (www.psrv.com) is a tool for debugging of Message Passing Interface (MPI)[6] programs. The debugger uses a monitor/replay approach, and allows the setting of breakpoints, instruction stepping and inspection of data-structures. The process of instrumentation, which is performed by a tool prior to compilation, can be parameterized to use different degrees of monitoring. Aspects that can be modified are different levels of loop profiling, external function profiling, I/O call profiling, and message passing profiling. During debugging, a lot of information can be gathered describing the balances of CPU usages, message send and message receive balances for individual nodes etc.

4.5.3 Hybrid Monitoring

According to Tsai et al. [62, s5.1;p104] hybrid monitoring come in two flavors, *memory-mapped*, and *coprocessor* monitoring. Memory-mapped monitoring uses a snooping device that listens to the bus, and reacts to operations on certain addresses. These addresses may either be snooping device registers that are memory-mapped into the address space of the task, or just a dedicated RAM area. Each event that should be monitored is forced to make a memory operation on the address that is associated with that event, which will allow the monitor to detect its occurrence. Coprocessor monitoring uses a device that is a coprocessor to the processor that executes the application that is to be monitored, events are forced to issue coprocessor instructions to the coprocessor as the events that are to be monitored will occur. The coprocessor monitoring approach requires, of course, that the architecture targeted allows the use of coprocessors.

The CodeTEST Trace Analysis tool from Applied Microsystems (www.amc.com)

[6]See www-unix.mcs.anl.gov/mpi/ for information about the MPI standard.

provides hardware assisted software based tracing of program execution. An extra stage is inserted into the compile stage where unique tags are added to the program code according to some parameters (thereby leaving the original code unchanged). A database is also created to relate the unique markers to specific lines of code.

Depending on where in the development stage the system is, different solutions are then used to collect information from the execution. Early in the design process a collection task that forwards the information to a remote host is run together with the normal task set; later in the process, tags are modified to only perform a memory read to a dedicated area, a hardware probe that can snoop the bus is then used to collect the information and send it to the remote host. Even though it is not intended to do so, at least the latter kind of probes may may be left in the system in order to avoid probe effect related problems.

The collected traces can be collected and viewed at three different abstraction levels: The high-level view shows task events and function entries/exits, thereby showing the context flow of the system. The control-flow view shows the execution path through the system. The source-level view displays each executed line of source code.

In "A Hardware and Software Monitor for High-Level System-on-Chip Verification" [49], El Shobaki and Lindh presents a method for monitoring SoC systems with a built in hardware component named MAMon (Multipurpose/Multiprocessor Application Monitor). The MAMon component is integrated with the design, and allows both hardware and hybrid monitoring. By using a hybrid approach, MAMon enables system level monitoring (see Section 4.5.2), while non-intrusive hardware monitoring can be used for The MAMon component can be used both with software based and hardware based [27] real-time operating systems. In the case where the operating system is hardware based, task information can be extracted non-intrusively from the kernel.

4.5.4 Discussion

A fourth type of monitoring would be above the level of software, to view the system as a closed box, and only monitor the effects that are visible to users of the system. Imagine a real-time control system, responsible for maintaining a level a certain level of the water in a cistern. To monitor the system from a level above the software could then be to monitor the water level, in order to evaluate the implementation of the control algorithm.

Noting that there are different levels of monitoring, and that each level have different advantages and drawbacks, we can state that monitoring at different levels is not strictly comparable. It is therefore likely that several levels of monitoring should be used, in order to obtain an overall picture of the system. But the choice of monitoring level is of course also dependent upon the bug-location hypothesis, and fault hypothesis.

If we for example assume a fault hypothesis that allows the potential presence of errors in the operating system, we must monitor the system on a low enough level, only probing individual tasks would not be sufficient. If we would like to recreate the execution of a nondeterministic program where all inputs are not available (see Section 2.4), or if parts of the log has been forfeited (see Section 4.3), we must make detailed recordings of the paths and data of the particular task. In such cases, we must be able to add probes into the application code.

Chapter 5

Reproducing the Execution of a Computer System

We have now provided a more detailed view of how a parallel system could be monitored, in this section we will probe the issue of execution reproduction in greater detail.

5.1 The Stampede Effect and the Bystander Effect

There are similarities not only between deadlock and livelock, or between the probe effect, the observability problem, and the probe-ability problem, but also between the stampede and bystander effects. Snelling and Hoffmann describes the two in their article “A Comparative Study of Libraries for Parallel Processing” published in 1988 [50]:

5.1.1 The Stampede Effect

As one task is forced to halt, by failure of execution or other reason, also all other tasks must be halted. If not, the other tasks may be able to corrupt data shared with the halted task. In the case of a failure, this will make it very hard to, by some form of postmortem analysis, find out exactly what happened.

We provide an example: Say that a task arrives too soon to a specific point in its execution. Because the task is early, assumptions about the state of shared resources that where made offline are not valid, and the state of the resource may have a state that designers assumed it couldn’t have. As the task uses the resource it eventually crashes, but the second task which is still alive, replaces the erroneous data with correct values before the whole system terminates. It is then impossible to, by viewing the memory state of the crashed system, determine what went wrong.

5.1.2 The Bystander Effect

The bystander effect also describes cases where tasks affect the state of others, but here the affected task terminates because other tasks are executing and violating some convention. Imagine that a failure occurs in a task, it will then seem probable that the cause of the problem resides inside that task. But either errors in the handling of virtual memory, or by infection through shared resources may cause a bystander to be affected by an error in a task that remains unaffected from its fault.

We provide an example which, strictly speaking, is not an example of the bystander effect, although it is confusingly similar to it. The example consists of two parallel Tasks T_A and T_B, see Figure 5.1 for the source of these: Say task T_A is supposed to receive an order for an action from task T_B, but T_B sends an invalid value. This will cause T_A to perform an unintended action, during which it may cause a failure, but T_B can proceed unaffected. However, in this case T_A is not innocent, a task is responsible for monitoring its own inputs (see the input errors of Clarke and McDermid at the top of Section 3). If however, the fault hypothesis for the system did not include the potential occurrence of input errors, this could be seen as a bystander error.

```
void f_1(void){printf("1");}
void f_2(void){printf("2");}
typedef void (*func_ptr)(void);
func_ptr fp[2]={f_1,f_2};
void A(void);
void B(void){
   send_to_proc(A,2);
}
void A(void){
   int i;
   i=recv_fr_proc(B);
   fp[i]();
}
```

Figure 5.1: This is Not an Example of the Bystander Effect

A more “pure” example of the bystander error would be if the data used by task T_A was modified without the tasks knowledge, if the task is not aware that it is receiving an external input, it cannot be held responsible for its inability to detect errors in that data.

5.1.3 Conclusion

When constructing a model for reproduction of an execution, care must be taken in order to guarantee that the stampede- and bystander effects are not allowed to show.

Both effects may show if the system is allowed to continue execution past a failure of a task without reporting this to the user. If the reproduction was not a success in terms of sensing an occurred failure, two problems may follow: Another, bystander task may then become infected. The traces of the failure may be erased by the execution of a stampeding “innocent” task.

This can become a reality if the reproduction mechanism has no clear sense of the system specification, but is also a potential problem during the monitoring activity, failure to log a change to a monitored entity may produce the same problems.

5.2 The Irreproducibility Effect and the Completeness Problem

The irreproducibility effect and the completeness problem are similar to each other in that both of them only emerge in nondeterministic[1] parallel programs [50] [18, s4.2.3;p87].

5.2.1 Irreproducibility Effect

The reproducibility problem, also known as the irreproducibility effect [18, 50], describes the fact that a certain behavior in a nondeterministic system cannot be repeated on command. Thus, it may be quite problematic to verify that a certain bug has been removed (see also Section 5.3 on regression testing), and also to distinguish between different bugs [18, s4.2.6;p94].

The irreproducibility effect is also referred to as the non-repeatibility effect [18, s4.2.6;p93].

Starting from a similar definition of deterministic systems as Kranzlmüller (see Section 2.4), and a definition for *Partial Determinism*, Thane [56, s3.2.2;p29] classifies systems with respect to their reproducibility:

A partially deterministic system has a certain behavior that can be defined by a known set of inputs or conditions, of which only a subset can be observed. A system is *Reproducible* provided that it is a deterministic system, and that all inputs that have impact on system performance are controllable. A system is said to be *Partially Reproducible* if it is deterministic, and a subset of the parameters that impact system performance are controllable.

[1]Sometimes also referred to as nondeterminacy or indeterminacy.

Note that, since it could never be determined that the reproduced execution is identical to the original execution, a reproduction of a partially deterministic system cannot be validated. To alleviate this problem it is imperative that the nondeterministic elements of the system are monitored, an issue which we discuss in Chapter 4.

5.2.2 The Completeness Problem

In order to ensure that a system complies to its specification it is required that the testing procedure is performed under realistic conditions. Properties that must be tested are both that the system reacts as intended on different input data, and (in the case of real-time systems) that the temporal behavior of the system satisfies the requirements. As different invocations of a nondeterministic program, per definition, can behave differently even though all controllable inputs are identical in all invocations, it is very difficult to determine the coverage of testing procedures. It is difficult to ensure completeness in the testing.

Testing the complete set of possible combinations of known input data and all execution orderings is normally referred to as exhaustive testing. Even in a very small system the number of test cases is very large, and it increases drastically as the system grows. Therefore, exhaustive testing is normally not an option as it would require too long time[2] to perform. The alternative is to only test a subset of the input combinations, which leads to that only a certain level of confidence may be ascribed to the systems capability to fulfill its specification. The level of confidence relates directly to how well the system was tested. It is true that small parts of the system, that are considered as especially important, could be selected for exhaustive testing. This would of course increase the confidence in the system, but is directly comparable to testing only a small subset of the possible input combinations.

Also, the completeness problem implies that even if the system would be tested with all possible combinations of inputs, bugs may still remain because different execution orderings in the system also affects the output and temporal behavior of the system. If the number of possible executions orderings are unknown, it may be difficult to determine the level of confidence that can be ascribed to the system. Thane et al. discuss this problem in [57, 59, 61] where they propose a method for testing real-time systems. The method describes how all possible orderings in a system can be identified, how all sequences of interleaving due to interrupts, blocking by semaphores, or scheduling decisions can be listed. They can then group a particular execution with an execution ordering. By running a sufficient number of tests and relating each test to its ordering, it is then possible to increase the confidence in the orderings that become subjected to testing. However, that simplified approach would either cause some of the less probable execution orderings to be insufficiently tested, or excessive testing due to the improbability or probability of experiencing those orderings. Therefore,

[2]Consider a program that subtracts one 32-bit integer from another, it would require $(2^{32})^2$ test cases. If one test case can be run each nano-second, that would result in $(2^{64} \cdot 10^{-9})/(60 * 60 * 24)$, or approximately 200'000 days of testing.

reproducibility in the testing is ensured by enforcing execution orderings during testing. By performing a sufficient amount of tests of a sufficient number of orderings, the confidence in the system can then be calculated based on the confidence in each ordering. In their articles, Thane et al. states that the number of execution orderings, and therefore also the testability of the system, is directly proportional to the number of preemption points and the jitter present in the system. Note that the confidence in a system according to Thane et al. can be a 2-dimensional property, a confidence in each execution ordering, and a confidence in covered execution orderings.

5.3 Regression Testing

As a bug is identified, and an attempt to remove it has been made, two things must be confirmed: (A.) The fix must not have introduced further bugs in the system. (B.) The bug must have been effectively removed. In deterministic systems, the process to confirm this is normally called *Regression Testing* [3], and it is performed by simply rerunning all previously performed tests after which the remaining tests can be performed.

However, in the case of nondeterministic systems, simply rerunning the previous test suite without errors does not prove any of the statements described in the above paragraph [3].

Carver and Tai propose [3] that this problem may be rectified by forcing deterministic executions according to given synchronization sequences. However, Thane and Hansson states [60] that a given execution trace of a program is only valid for an altered version of that program if the alteration does not affect the execution, which implies that the regression testing procedure cannot make use of pre-bugfix recorded logs.

Neri et al. elaborates further on the problem in “Debugging Distributed Applications with Replay Capabilities” published in 1997 [33], they point out several practical problems with reusing of monitoring logs. If an executable is modified, either by re-compilation or re-linking (note that it is not required that the code is changed, different options to linkers *etcetera* may accomplish the same problem), address references may be changed. Therefore, in order to alleviate this problem, they propose that check-sums of binaries should be calculated, and that these should be added to the log, in order to detect the problem. Also the use a virtual memory and caching schemes requires some thought, as physical addresses may change between executions, causing differing behavior in the caches if initial memory states are not identical. This could result in that two executions, that in all other aspects are identical, may have differing logs.

Thus, we conclude that the area of regression testing of parallel systems needs further research.

5.4 Uses of Monitoring Output

McDowell and Helmbold [30] stated four different uses of execution logs, Browsing, Replay, and Simulation. Which method that can be used depends on how much information that is logged from the reference execution.

5.4.1 Browsing

By viewing the recorded history in a very simplified model of the target platform. When browsing event histories, it may even be possible to use the same model for different architectures. The programmer can observe the ordering of events in the system, and draw conclusions from that. The perhaps most significant advantage of this approach is that it allows a large level of abstraction from the sometimes too detailed view normally provided in traditional debuggers [30].

The MAMon monitoring component for SoC systems presented by Shobaki and Lindh [49] is one example of an approach that uses browsing of event histories to display monitoring output.

5.4.2 Replay

A new, *replay*, execution is performed on the target environment, but the replay execution is forced to correspond to the original reference execution. The programmer is therefore allowed to stop the system, even to stop only some of the system entities, because the replay mechanism will not allow the replay to violate constraints derived from the reference execution.

Kilgore and Chase presents a method in “Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages”, published in 1997, [15] which is targeted at message passing systems that are *piece-wise deterministic*. They define a piece-wise deterministic system to be a system whose only element of nondeterminism is the ordering of message deliveries.[3] In other words, given two instances of the same program executions, provided that all messages are delivered in the same order to both instances, the two will be identical. The Kilgore and Chase approach identifies possible data races in a program execution, and can then, according to some rules, reorder the sent messages with the intention to provoke a failure.

Russinovich and Cogswell present a method that facilitates deterministic replay on nondeterministic shared-memory uni-processor systems in “Replay for Concurrent Non-Deterministic Shared-Memory Applications” (1996) [47]. The approach is called *repeatable scheduling algorithm*, and it ensures deterministic replay by forcing the system to make the same scheduling decisions during replay as during the reference execution. In order to do so, it requires the use of Software Instruction Counters. If the initial state of the reference and the

[3]However, it seems reasonable that also the timing of the message deliverances can have impact on system performance, especially in real-time systems, but also in other systems.

replay executions are identical, this will guarantee that the two executions are identical. Note that this method is not sufficient in systems with more than one processor, or in systems that take input from external processes. Both these limitations gravely reduces the applicability of the approach, but it can of course be used in conjunction with other methods.

Lumpp et al. stresses the fact there are other issues than errors in parallel systems that may profit from the parallel debugging methodologies. Because dynamic methods that facilitate replay in these systems will also provide detailed knowledge on low-level system functionality, they can also be used for *performance debugging* [28]. They present a debugger for distributed shared memory systems. Suárez et al. [55] also presents work in the area of performance debugging, they are targeted at distributed embedded real-time systems.

Boothe presents a method for bidirectional stepping through sequential code in "Efficient Algorithms for Bidirectional Debugging" [2]. By monitoring using Software Instruction Counters (see Section 4.2), and also counting the function entering and exit (there may be several different exit points from a function) points, executions logs are created. The logs will contain sufficient information to facilitate execution reproduction, and and also to identify individual instructions. Breakpoints are specified as counter configurations. As individual instructions can be identified in an orderly fashion, the debugger can also also perform backwards stepping. If the counters are set to indicate the previous instruction, and the program is re-executed, this will create the illusion that the program is being stepped backwards.

5.4.3 Simulation

By using a simulator of the target system, and forcing it to behave in a way that will produce an replay execution that is identical to the reference execution, the programmer can make repeated executions of the system.

This requires either that the model used, the simulator, models the real target system sufficiently accurate for the application, or that it can be forced to execute the system according to the traces recorded previously during the reference execution. As the simulator will execute the same code as that which was run during the reference execution (see Section 5.3), we can state that the above stated requirement "sufficiently accurate" would be satisfied when the log produced during the simulation does not deviate from the reference log.

In "Using Deterministic Replay for Debugging of Distributed Real-Time Systems" [60] Thane and Hansson describes the, to our knowledge first, method for deterministic replay of distributed real-time systems. The method is based on an operating system that provides monitoring primitives for task level monitoring, and that also monitors its internal event sequences. They use a software monitoring approach, and avoid the probe effect by leaving probes in the system. For the ordering of events, they assume that the system provides a synchronized global time-base.

5.5 Visualizing the Debugging Process

As stated by Kranzlmüller [18, s4.2.1;p84] it is very important that the programmer can understand the context of the debugging process, what is being debugged at a certain time. However, in large systems, especially in cases where the compiler has used optimization techniques, this may be rather difficult.

The systems that we target in our work are rather complex, meaning that a lot of information about their current state is required in order to fully understand what is happening. This is of course a relative measure. In order to put a bit more perspective on the issue we can add that all information needed to solve the task of efficiently debugging the system should be readily displayed on a normal computer screen. In addition it must be so in such a fashion that a programmer can understand and use the information displayed without feeling that he or she is compromised by the interface.

This is a potentially large problem in this type of systems, we must find new means of refining, distilling, and displaying, information to the programmer.

McDowell and Helmbold [30] presented four means of presenting this information. Also Pancake and Utter have done some work in the area [39].

Chapter 6

Future Work

After that a more comprehensive historical investigation has been completed, when we have surveyed actual solution proposals to the sketched methods presented here, we will commence work on one or more of the topics presented in this section.

6.1 Deterministic Replay

We will in our future work concentrate on the simulated deterministic replay approach, using software monitoring, it seems that there are some issues that require investigation.

External devices in simulated replay is an issue that has not been investigated; how can we debug a system that uses a hard disk, possibly even a swapping algorithm? This points at a problem that is inherent in the simulation approach, namely that a simulated machine does not always behave in the same way that a real system would. Reasons to this are varying. In some cases, simplifications of the model are judged not to give great impact on performance. In other cases, it is not possible to build a model that behaves exactly as the original component.

An issue that is constantly present in deterministic replay, but is aggravated when we target more complex parallel systems, is that of the amount of data produced by a monitoring mechanism. As the amount of data that is needed per time unit grows, this may also affect the system performance, thereby reducing the use of the method. A checkpointing system could reduce the amount of data needed to perform the replay of the system, but would consume resources from the system during run-time. How to perform these checkpointing operations so that their impact on performance is minimized, and keep consistency in the monitoring traces is an important issue in the context. Another approach is to accept that some of the collected data will be lost, and adopt to that fact. Browsing (see Section 5.4.1) as method of replay would perhaps not suffer as much from this approach as replay and simulation (see Sections 5.4.2 and

5.4.3). To which extent we could perform these, under these restrictions, more complicated methods of replay, is an interesting topic.

Furthermore, there are other inherent issues of the simulated replay approach that could be improved. The simulation of parallel architectures enforces a large slowdown, simulating software takes in the order of hundred, or even thousand, times as long as native execution [10, s4.4.4;p58]. In other cases, this is an overhead that one must learn to live with, but in the case of simulated replay, we have additional information about the execution that may help us to reduce that overhead.

In Section 4.3 we motivated the need of a well defined starting point when using simulated replay. If we are to make effective use of the deterministic simulated replay methodology on parallel architectures, we must determine how we may find such a starting point when the simulation has proceeded long enough to have overwritten part of the gathered log. In Section 4.5.4, we imply that we may view such systems as nondeterministic or partially deterministic (see Section 5.2.1). The loss of some of the information that defines the execution may satisfy the rules for nondeterministic systems (see Section 2.4) if there is not a sufficiently large amount of task level (see Section 4.5.2) traces, in which case it may satisfy the criteria for a partially deterministic system. Whether they are reproducible, or partially reproducible (see Section 2.4) remains to be seen.

6.2 Debugging Component Based Systems

In Section 4.5.4, we saw that some systems require that monitoring is performed also on task level, that control and data flow inside individual tasks must, in some cases, be monitored. This requires that the source code of those tasks is available and possible to modify, such is not always the case in Component Based Software Engineering (CBSE). However, we in such systems we may insert probes into the code that uses the component(s), and if we have control over the operating system, we can monitor the system on that level.

An interesting question is to what extent such systems may be observed and replayed; is it possible to find all bugs inside the code that is available for change, and is it possible to identify faulty components?

If the bug resides inside a component, it is desirable to be able to describe the situation that produced a failure to the vendors of the component. In order to do so we should record all interaction sequences between the user of the component and the component, but has the same problem with long executions as described in Section 4.3.

It is, of course, possible to build components that have built-in monitoring facilities. But this requires either very extensive monitoring, by the user adjustable monitoring (which is difficult due to the probe effect), or very detailed comprehension of how the component is used in a special case. As one of the major gains of CBSE is increased reuse, and users want to use the components in slightly differing contexts, it may be difficult not to do over enthusiastic monitoring if the level of monitoring granularity is static.

6.3 Design Patterns for Design of Probe-able Systems

As we have pointed out in this report, an inherent problem with monitoring computer systems is the costs. These costs can be measured both in a temporal and in a spatial dimension, and it is an implementation specific choice in which dimension to optimize the behavior of the monitoring mechanism.

We believe it possible to find some general rules that, should they be acknowledged in the system design, can reduce the monitoring-enforced penalty in one of these dimensions.

6.3.1 Sketched Examples of Design Patterns

For example, these rules could restrict the spatial scattering of data that is to be monitored. In a system, the meaning of a task instance should be defined to facilitate Incremental Replay (see Section 4.5.2 and [37, 65] on Incremental Replay). Between each such task instance, at least all data which cannot be reproduced must be monitored and stored. If all this data is stored in a easy to reach structure, this would certainly ease the monitoring effort.

Other rules could restrict the use (and re-use) of temporary variables, so that they could be excluded from the subset of monitored variables. We note that a variable that potentially has scope between two iterations of a task must be monitored in order to allow the independent recreation of a particular instance. If a temporary variable is allowed a greater scope then necessary, or if the same variable is re-used in independent operations, this can lead to an increased need of monitoring that really could be optimized.

Yet other rules could assist in reducing the jitter in the system. The presence and span of jitter in a system increases non-determinism, and therefore also the potential for race conditions. Should the amount of jitter be reduced, this would reduce the number of entries in the control-flow monitoring without requiring individual entries to be larger. The reasons for jitter in a system are many, ranging from accumulating effects due to inter-task dependencies, to varying execution times due to non-determinism in selections. Ways to reduce jitter in a system should therefore also be many, some could aim to reduce the amount of selections in the system, others to shorten the chains of inter-task dependencies.

6.4 Comparing Tools for Debugging

As we have seen in this report, there exist a couple of different tools that can be used when debugging. We have also seen that there are some costs involved when using these tools. What we have not seen is a comparison between the tools, we have not which tool is the most efficient in some relevant aspect.

The reason for this insufficiency is that the existing implementations have been made on different, incomparable, platforms. Thus, a strict comparison is not feasible, other means of comparison must be made. In their article “A Taxonomy of Distributed Debuggers Based on Execution Replay” [7], Dionne et al. present a taxonomy which can be used to classify debuggers with respect to nine (9) criterion’s. This, together with a fault hypothesis, can be used to choose a tool suitable for a given project.

However, the presented taxonomy does not cover any real-time aspects. There are also other insufficiencies, one of these being the way the probe effect is handled; Schütz [48] states three classes based on how they handle the probe effect: by ignoring the effect, by minimizing the impact on the system during debugging, or by avoiding the probe effect. Other insufficiencies are in the range of solution alternatives in surveyed topics: Integration of probes to the system is said to be possible by automatic- (complete or partial) or manual insertion, where manual insertion is tailor made for a particular system, and automatic is performed with a tool. As we have seen in this report, also other methods are possible (see Section 4.5.2 and the kernel probes suggested by Thane which are integrated manually but also reusable).

Furthermore, the range of tools which have been mapped with the taxonomy is small. In future work, we plan to remedy this and also to extend the taxonomy.

6.5 Efficient Memory Usage in Storing Monitoring Entries

Stewart and Gentleman [53] mentioned the applicability of circular queues as an infrastructure when storing monitoring entries. It seems that the potential for keeping redundant information in such a scheme is larger then needed. This was also implied by Ronsse and De Bosschere [43], they stated that entries should be evicted as soon as they are without use.

When using a circular queue structure, garbage collection is trivial; entries can be stored in chronological order on the medium, and as space is exhausted the oldest entry is replaced with the newest entry. Thus, the on-line performance of the garbage collection algorithm ensures that no large penalty is imposed on the system.

However, there are other performance related drawbacks to this simplistic scheme. These issues do not concern the on-line performance of the algorithm, but the off-line usefulness-ratio of the stored information. That is to say how many of the stored entries that can be used in a replay. In the circular queue solution, no respect is paid to the relative context of the information which is expunged and the information which is allowed to remain. The usefulness of the information handled is ignored. Thus, we cannot assume that the final product is optimal with respect to the off-line usefulness of stored data.

Furthermore, out of a complexity perspective for the programmer, it is desirable to allow replay of only a subset of the system. As only a subset of the system is replayed, only that subset must be monitored - thereby requiring less of the limited memory resources. But, as mentioned in Section 4.1.1, probes should not be removed from, or added to, the system because it invalidates previous verification efforts. But, if the functionality of the garbage collection algorithm could be altered without introducing a probe effect, memory resources could be saved.

We intend to develop a new infrastructure for storing of monitoring activities. The intention of that work should be to reduce the amount of unusable information which accumulated in the monitoring-log.

6.6 Conferences and Research Groups of Interest

Forums in which future results in the field of debugging of parallel systems may be published include several groups. Some results have been published in real-time forums, others have been published in the distributed and parallel systems community. But there are also channels primarily dedicated to distribute results in the domain of testing and debugging of computer systems.

Examples of conferences are IEEE Parallel and Distributed Systems, IEEE Symposium on Reliable Distributed Systems, ACM International Symposium on Software Testing and Analysis.

Among the research groups and their projects that are currently active in the field, we mention the following:

TUM at the Fakultät für Informatik of the Technische Universität München, there is a group that does work in programming development environment and tools. Their homepage is located at wwwbode.cs.tum.edu/Par/tools/-index.html.

Johannes Kepler Universität in Linz, Austria, has a group at the Department for Graphics and Parallel Processing. The group has a project that deals with the debugging of distributed memory machines, a project homepage is available at www.gup.uni-linz.ac.at/research/debugging/-index.php

The Australian National University in collaboration with Fujitsu Laboratories Ltd. has a rather extensive research program called CAP which has published some work in the area of debugging parallel computers. The homepage of the program is available at cap.anu.edu.au/.

PARIS in the Department of Electronics and Information Systems at Universiteit Gent, Belgium, has a group lead by Koen De Bosschere that have active research in the field of debugging parallel programs. The PARIS group has a homepage at www.elis.rug.ac.be/ELISgroups/-paris/index.html, and they describe a project called RecPlay at sunmp.elis.rug.ac.be/recplay/

Chapter 7

Summary

We have in this report surveyed the different problems that exists in debugging of parallel applications, and different effects that influence parallel program execution. A successful approach to debugging must direct all of these, or suffer from limited applicability. We have described why the classic cyclic debugging approach cannot be used as-is on parallel systems, and we have given an introduction to replay which can facilitate the use of cyclic debugging in these systems. As there are several approaches to perform the monitoring required by the replay, we have also briefly described the main approaches to do this.

Of the different papers that were read during this work, the following are perhaps more important than others:

Schütz [48] provides a very comprehensive survey of the research area of testing distributed real-time systems up until 1994.

We note that McDowell and Helmbold provided a comprehensive summary of the area of parallel debugging in their now classic paper on parallel debugging [30]. They explain many of the general problems that are encountered when trying to debug parallel programs, and also provide some views on the different solutions available. This paper gives a very good introduction to the field.

The probe effect was first named by Gait in "A Probe Effect in Concurrent Programs" published in 1986 [9]. However, LeDoux and Parker have previously mentioned the phenomenon in "Saving Traces for Ada Debugging" [24], but referred to it as Heisenbergs Uncertainty principle.[1]

Among recent dissertations in the field, we mention Henrik Thane [56] (2000), and Dieter Kranzlmuller [18] (2000).

[1]As the first draft of Gaits paper was received by the review committee in late 1984, we can not say for sure which of the two groups that actually thought of the problem first. It may even be someone completely different who deserves the credit.

Bibliography

[1] Anish Arora et al. Resettable Vector Clocks. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 269 – 278, July 2000.

[2] Bob Boothe. Efficient Algorithms for Bidirectional Debugging. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, volume 35(5) of *SIGPLAN Notices*, pages 299 – 310. ACM, May 2000.

[3] Richard Carver and Kuo-Chung Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66 – 74, March 1991.

[4] Randy Chow and Theodore Johnson. *Distrubuted Operating Systems & Algorithms.* Addisson Wesley Longman Inc., 1997.

[5] Stephen Clarke and John McDermid. Software Fault Trees and Weakest Preconditions: A Comparison and Analysis. *Software Engineering Journal*, 8(4):225 – 236, July 1993.

[6] Edward Coffman et al. System Deadlocks. *Computing Surveys*, 3(2):67 – 78, June 1971.

[7] Carl Dionne et al. A Taxonomy of Distributed Debuggers Based on Execution Replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 203 – 214, August 1996.

[8] Colin Fidge. Fundamentals of Distributed System Observation. *IEEE Software*, 13(6):77 – 83, November 1996.

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

[10] Stephen Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior.* PhD thesis, Stanford University, USA, February 1998.

[11] Scott Howard. A Backgroud Debugging Mode Driver Package for Modular Microcontrollers. Technical Report Motorola Semiconductor Application Note AN1230/D, Motorola Inc., 1996. http://e-www.motorola.com/brdata/PDFDB/docs/AN1230.pdf.

[12] IEEE. *IEEE Standards for Local Area Networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.* 1985. IEEE Std. 802.3-1985.

[13] IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture.* 2001. IEEE Std. 1149.1-2001.

[14] Pete Keleher et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115 – 131, January 1994.

[15] Richard Kilgore and Craig Chase. Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 1, pages 423 – 432, Januari 1997.

[16] Harry Koehnemann and Timothy Lindquist. Towards Target-Level Testing and Debugging Tools for Embedded Software. In *Conference Proceedings on TRI-Ada*, pages 288 – 298. ACM, September 1993.

[17] Hermann Kopetz and Wilhelm Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *Transactions on Computers*, 36(8):933 – 940, August 1987.

[18] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johannes Kepler University of Linz, Austria, September 2000.

[19] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558 – 565, 1978.

[20] Leslie Lamport et al. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382 – 401, July 1982.

[21] Jean-Claude Laprie. *Dependability: Basic Concepts and Associated Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1992.

[22] Thomas LeBlanc. Parallel Program Debugging. In *Proceedings of the 13th Annual Computer Software and Applications Conference COMPSAC'89*, pages 65 – 66, September 1989.

[23] Thomas LeBlanc and John Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *Transactions on Computers*, 36(4):471 – 482, April 1987.

[24] Carol LeDoux and Stott Parker. Saving Traces for Ada Debugging. In *Proceedings of the Ada International Conference on Ada in Use*, pages 97 – 108. ACM, May 1985.

[25] Nancy Leveson. *Safeware - System, Safety and Computers*. Addison Wesley, 1995.

[26] Luk Levrouw et al. A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks. In *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, pages 471 – 478, Januari 1994.

[27] Lennart Lindh et al. Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems. In *the Seventh Swedish Workshop on Computer Systems Architecture*, June 1998.

[28] James Lumpp et al. Xunify - a Performance Debugger for a Distributed Shared Memory System. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 587 – 596. IEEE, Januari 1998.

[29] Ciaran MacNamee and Donal Heffernan. Emerging On-Chip Debugging Techniques for Real-Time Embedded Systems. *Computing & Control Engineering Journal*, 11(6):295 – 303, December 2000.

[30] Charles McDowell and David Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593 – 622, December 1989.

[31] John Mellor-Crummey and Thomas LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86. ACM, April 1989.

[32] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[33] Daniel Neri et al. Debugging Distributed Applications with Replay Capabilities. In *Proceedings of the 1997 conference on TRI-Ada*, pages 189 – 195, November 1997.

[34] Robert Netzer. *Race Condition Detection for Debugging Shared-Memory Programs.* PhD thesis, University of Wisconsin, USA, August 1991.

[35] Robert Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28(12) of *SIGPLAN Notices*, pages 1 – 11. ACM, December 1993.

[36] Robert Netzer. Trace Size vs Parallelism in Trace-and-Replay Debugging of Shared-Memory Programs. Technical Report CS-93-27, Department of Computer Science at Brown University, June 1993.

[37] Robert Netzer et al. Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 404 – 413. IEEE, June 1994.

[38] Robert Netzer and Barton Miller. What are Race Conditions? - Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74 – 88, March 1992.

[39] Cherri Pancake and Sue Utter. Models for Visualization in Parallel Debuggers. In *Proceedings of the 1989 Conference on Supercomputing*, pages 627 – 636. ACM, November 1989.

[40] Bernhard Plattner. Real-Time Execution Monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756 – 764, November 1984.

[41] Stefan Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems.* PhD thesis, Technishe Universität Wien, Austria, April 1994.

[42] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *Transactions on Computer Systems*, 17(2):133 – 152, May 1999.

[43] Michiel Ronsse and Koen De Bosschere. Non-Intrusive On-the-Fly Data Race Detection Using Execution Replay. In *Fourth International Workshop on Automated Debugging*, pages 148 – 163, August 2000.

[44] Michiel Ronsse et al. Execution Replay and Debugging. In *Proceedings of the Fourth International Workshop on Automated Debugging*, pages 5 – 18, August 2000.

[45] Michiel Ronsse et al. Cyclic Debugging Using Execution Replay. In *International Conference on Computational Science*, volume 2074 of *LNCS*, pages 851 – 860, May 2001.

[46] Michiel Ronsse and Willy Zwaenepoel. Execution Replay for TreadMarks. In *Proceedings of the Fifth Euromicro Workshop on Parallel and Distributed Processing*, pages 343 – 350, January 1997.

[47] Mark Russinovich and Bryce Cogswell. Replay for Concurrent Non-Deterministic Shared-Memory Applications. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, volume 31(5) of *SIGPLAN Notices*, pages 258 – 266, May 1996.

[48] Werner Schütz. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems*, 7(2):129 – 157, September 1994.

[49] Mohammed El Shobaki and Lennart Lindh. A Hardware and Software Monitor for High-Level System-on-Chip Verification. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pages 56 – 61, March 2001.

[50] David Snelling and Geerd-R. Hoffmann. A Comparative Study of Libraries for Parallel Processing. *Parallel Computing*, 8(1-3):255 – 266, 1988.

[51] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice-Hall Inc., 2001.

[52] IEEE Industry Standards and Technology Orginization. *The Nexus 5001 Forum Standard for a Global Embedded Debug Interface.* 1999. IEEE-ISTO 5001 1999.

[53] Darlene Stewart and Morven Gentleman. Non-Stop Monitoring and Debugging on Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 263 – 269. IEEE Computer Society, May 1997.

[54] Robert Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204 – 226, August 1985.

[55] Francisco Suárez et al. Performance Debugging of Parallel and Distributed Embedded Systems. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 135 – 149. IEEE, 2000.

[56] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems.* PhD thesis, Kungliga Tekniska Högskolan, Sweden, May 2000.

[57] Henrik Thane et al. Integration Testing of Fixed Priority Scheduled Real-Time Systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, December 2001.

[58] Henrik Thane et al. The Asterix Real-Time Kernel. In *Proceedings of the 13th Euromicro International Conference On Real-Time Systems*, June 2001.

[59] Henrik Thane and Hans Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of the 20th Real-Time System Symposium*, pages 360 – 369. IEEE, December 1999.

[60] Henrik Thane and Hans Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *the 12th Euromicro Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.

[61] Henrik Thane and Hans Hansson. Testing Distributed Real-Time Systems. *Journal of Microprocessors and Microsystems*, 24:463 – 478, February 2001.

[62] Jeffrey Tsai et al. *Distributed Real-Time Systems: Monitoring Visualization and Debugging and Analysis.* Wiley-Interscience, 1996.

[63] Yi-Min Wang and Kent Fuchs. Optimal Message Log Reclamation for Uncoordinated Checkpointing. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 24 – 29. IEEE, June 1995.

[64] David L. Weaver and Tom Germand, editors. *The SPARC Architecture Manual.* PTR Prentice-Hall, 1994.

[65] Franco Zambonelli and Robert Netzer. An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications. In *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 392 – 398. IEEE, April 1999.