Mälardalen University Licentiate Thesis
No.16

Preparing for Replay

Joel Huselius

November 2003

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 91-88834-15-8
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Cyclic debugging is the process normally used for examining and removing bugs in computer systems. For this process, the possibility to deterministically repeat executions is a requirement - without repeatible experiments, it is not certain that existing bugs can be located. Thus, in order to debug real-time systems, which normally do not allow repeatable experiments, additional methods are needed to provide repeatability. Several solutions based on a resource demanding record/replay approach have been proposed: By recording data describing the occurencies of non-deterministic events during a reference execution, and then using this data to force a consequtive replay excecution to perform in the same way as the reference, repeatability in experiments is achieved.

We adhere to the previous work on deterministic replay by Thane et al. The method assumes that memory resources have limited capacity compared to the amount of data recorded. This assumption leads to that data available after the completion of the reference execution does not cover the reference execution in its entirety, wherefore replay must be started from a state which is not the initial state of the system. To facilitate this, at predefined locations in the code, checkpoints are taken of the individual task-states. In order to reduce the overhead imposed on the system, checkpoints are not required to be exhaustive, only to cover the parts of the data-space with non-deterministic properties.

The combination of these factors leads to an environment that requires new methods for initiating replay - one of the contribution of this thesis is such a method. By treating each task in the system independently, we show (by means of an industrial case-study) that a restarted version of the system can be made to look like the reference execution.

In order to guarantee that a replay execution can always be performed, the addition of this new method triggers the requirement of new dynamic methods for managing data during recording. The second contribution of this thesis is a dynamic memory manager that fills this gap and is also shown to improve memory utilization in sporadic real-time systems.

Abstract

Cyklisk debuggning är en vanlig process för att debugga datorsystem. Processen kräver ofta repeterade experiment (därav namnet), vilket endast är möjligt om exekveringen av systemet är möjligt att reproducera - eftersom att parallella och/eller realtidssystem normalt inte kan garantera detta krävs en speciell lösning. Record/replay är en ofta använd sådan, vilken i stort fungerar som en videobandspelare; genom att först spela in en referensexekvering av systemet kan denna sedan deterministiskt återskapas om och om igen i en modell av systemplattformen.

En stor nackdel med denna lösning är att resursåtgången för att spela in en exekvering är stor - både vad gäller minnes- och tidsåtgång. Tidigare lösningar har använt checkpoints för att slippa spara inspelningar som sträcker sig över hela systemets exekveringstid (vilken kan vara dagar, till och med år). Vi har i detta arbete vidareutvecklat en lösning av Thane et al. vilken använder icke-kompletta checkpoints för att ytterligare minska minnesåtgången vid inspelning. Genom att utesluta de deterministiska elementen från checkpoints så kan resurser sparas. Dock, användandet av sådana checkpoints leder till ett behov av nya metoder som kan starta exekveringen av systemet i ett tillstånd som inte är systemets initialtillstånd.

I denna uppsats presenterar vi en sådan metod vilken fungerar genom att behandla var task (eller process) för sig; genom en case-study utförd i industrin visar vi att metoden verkligen kan få ett återstartat system att se ut som referensexekveringen. Det visar sig dock att användandet av metoden kräver nya dynamiska metoder för att hantera minnesutrymmet allokerat för data inspelat under referensexekveringen. Utan en sådan metod kan inga garantier ges för att exekveringen kan återskapas. Vidare krävs av denna metod att den har en konstant exekveringstid; utan konstant exekveringstid kommer jittret i systemet att öka vilket leder till att systemet blir mer yttermera svårt att testa. Vi presenterar även en sådan metod, med konstant exekveringstid, och visar i en utvärdering att den, förutom att som första metod fylla samtliga av de uppställda kraven på en sådan metod, även ger ett bättre resultat i sporadiska realtidssystem än tidigare kända metoder.

Till Rebecca

Acknowledgements

This work has been supported by the Swedish Foundation for Strategic Research (SSF) via the research programme SAVE, the Swedish Institute of Computer Science (SICS), and Mälardalen University.

On a personal note, I would to thank everybody at the department, great people, the lot of you! Special thanks goes to my fellow PhD students, especially Daniel Sundmark Anders Pettersson with whom I share project and room, Dag Nyström accompanied by Thomas Nolte and Jonas Neander from my time as an undergraduate. Many thank-you's goes to Jocke Adomat, Professor Christer Nordstöm, and Filip Sebek for the briljant undergraduate courses that made me understand how much I did not understand - and made me want to learn more. Also the administrative staff at the department, for practical issues, happy faces, and positive attitudes; Harriet, Monica, Malin... During my time at the department, first as a master student and then as a PhD student, Professor Gerhard Fohler has been a supportive, encouraging, and understanding friend/MSc-thesis supervisor. My two current supervisors Dr Henrik Thane and Professor Hans Hansson have of course provided solid support, ideas, (mostly constructive) criticism, and hard work, all vital to the results here presented. Also Lars Albertsson from SICS has had significant influence on the work presented. I would like to thank Dr. Andreas Ermedahl for reading and producing some good comments on the final draft of this thesis.

Further, there is also a great number of people from my life outside the department that have supported me during these years (and, in some cases, much longer), these various people well deserve my humble appreciation: My relatives (in no particular order, just everybody, past, present, and future generations, always). My most valued travelling companion, my every day motivation, the better half of my family, and the love of my life; Rebecca. Mats, Gunnel, Mats, Ingrid - thank you! Longtime-hardcore-friends, equal coffee-persons, fellow skiers, school mates (at different points in time), infamous truants and accomplices, recreational drinking companions, and general role-models: Gustav and Malin, Pudel-Peter, Jocke and his concrete table, no-longer-snoring Stefan, Danny (Assian-assar-osser), Ingvar with the three dwarfs, Sjöbusen Emma the Pirate, Linus the virus, Johnnie B Good, Micke

Bensin, Mannelito, Krasse, Cribbe, Jonas the Neanderthal, Johan, Anneke, Pernilla, Kimmo, Johan, Peter, Stefan, Göthe... Thank you for all the good times!

Last, but not least, I would also like to thank everybody that has put their valued time and hard effort into non-profit work for the LTD (Linjeföreningen för Tillämpad Datateknik) fraternity; by doing what they do best, these people have made studying computer engineering (and the associated activities) at Mälardalen University a lot more fun since approximately 1989. In this way, as I acted as vice chairman of LTD in 1998, and consultative member of the board in 1999, I thank also my self... - Go Joel, Go Joel!

Thank you, all!

Bromma, October 2003
Joel Gustaf Huselius

Contents

Chapter 1

Introduction

The research described in this thesis[1] concerns debugging of real-time systems. Normally, debugging (or *cyclic debugging*) is an iterative process (hence "cyclic") performed by setting breakpoints and stepping through the system source code over and over again, inspecting program state as you go. This approach, however, is not directly applicable to non-deterministic and/or time-dependent systems (e.g. real-time systems).

With respect to debugging, two things differentiate real-time systems from those that can be cyclically debugged: Firstly, because of non-deterministic races and irreproducible input from the surrounding environment, it cannot be ensured that two consecutive executions will be identical. Secondly, perturbing the system will alter its behavior (hence, no breakpoints etc. may be used on the system as-is) - this is known as the *probe effect* [4].

The solution that we and many others [1, 5, 11, 19] use is the record/replay approach to facilitate cyclical debugging. The process consists of two steps: first, a *reference execution* of the system is executed and observed, second, a *replay execution* is performed based on the observations made during the reference execution. This consecutive replay execution is repeatable (as long as the observations made from the replay execution are available), and is not vulnerable to the probe effect (as the decisions that may be effected are controlled by the observations made from the reference execution). Thus, the replay execution can be debugged using conventional methods, and as long as it can be considered to be a facsimile of the reference execution, debugging the

[1] You are reading a thesis presented in candidacy for the degree of Technology Licentiate; a Licentiate is a swedish graduate degree ranked between Master and Doctor.

replay execution is the same as debugging the reference execution.

The act of observation is referred to as *recording* the reference execution. Recording has two sub-activities: First, *monitoring* the system, second, *logging* monitoring data. Monitoring is performed by inserting *probes* into the system, in order to extract *entries* with information about the execution. Logging is the act of saving these entries into *records*, and managing the space available for that process. In this work, we assume that logging is performed locally; data is not stored on some offline node with unlimited capacity. As a result of this assumption, there is a competition between log-entries for storage space. We therefore require that there is an algorithm that can prioritize over the available entries so that the most valuable (in some appropriate respect, see Paper C) entries are kept. A *logging structure* is the abstract data type (ADT) responsible for the online management of storage space, the structure can be compared to a garbage collection algorithm in the sense that it tries to identify previously used space to allocate for new data.

Probes can be implemented in software, hardware, or some hybrid. The difference between solutions can essentially be compared by analyzing the perturbation on the system incurred by the probes, but there is also (for example) an economical cost issue. Generally, hardware implementations have low perturbation, low abstraction level, and high economical cost, while software implementations have opposite characteristics. If probes are perturbing the system, they cannot be removed, altered, or added, without modifying the conditions for the remaining system. If these conditions are modified, a *probe effect* will be incurred on the system [4], resulting in that previous system validation efforts are made obsolete. In this work, we assume that probes are perturbing and implemented in software.

Note that, in the interest of testability, the perturbation incurred on the system by these probes should be constant with respect to time; the recording of a given entry should always consume the same resources. Jitter in the probes will increase the jitter of the application, which will decrease testability [15].

1.1 Problem formulation

In this work, we address the general issue of debugging real-time systems. This is assumed to be performed with a record/replay solution that can remedy the inherent problems of cyclically debugging non-deterministic and/or time-dependent systems. In order to minimize the system perturbation (with respect to the execution overhead) of the approach, we use *memory excluding*

checkpoints [8] (see Section 1.2.3).

More specifically, we are investigating two sub-issues: first, how to use available space for logging data from the monitoring process, second, how to start a replay execution from another state than the initial system state when using memory excluding checkpoints. We show that these issues are related (Paper B); the algorithm for storing data from the monitoring process can guarantee that a replay of a system is always feasible. For the first issue, we present a novel algorithm (Paper E), the successor of which is presented as a contribution in this thesis (Paper C). For the second issue, we investigate the system constraints and present a method for how to perform this start-up of the replay-system (Paper B).

1.2 Related work

With respect to related work in the field of replay debugging of concurrent programs and real-time systems, most references are quite old. Recent advancement in the field has been meagre. On the special topic of finding starting points for replay of real-time systems, no comprehensive studies have been published hitherto. The only work known to us that has some similarities [7, 19] is limited to replay of message passing in concurrent software, and does not cover real-time issues like scheduled preemptions, access to critical sections, or interrupts. Also, the jitter of these solutions cause the testability to be compromised.

A more comprehensive study of related works than found in this section is provided in Paper A.

1.2.1 Replay

On the general topic of replay, much of the previous work published has either been relying on special hardware [3, 17] or on special compilers generating dedicated instrumented code [3, 6]. This has limited the applicability of these solutions on standard hardware and standard real-time operating system software.

Other approaches do not rely on special compilers or hardware, but lack in the respect that they can only replay concurrent program execution events like rendezvous, but not real-time specific events like scheduled preemptions, asynchronous interrupts or mutual exclusion operations [1, 11, 19].

Xu et al. [18] present their Flight Data Recorder (FDR), an intrusive

hardware that facilitates debugging of races in multiprocessor systems. Similair to our work (see Paper F), the intention is to leave the facility in the deployed system, thereby proividing a powerfull tool for that aid system maintainance. FDR allows a replay to be started from a state which is not the initial state of the system, but require complete system checkpoints as starting point. In order to reduce the system perturbation from checkpointing, incremental checkpoints are used in the described implementation.

1.2.2 Jitter

A *jitter* is a difference - for example in the execution time of an algorithm. Say that the fastest execution of a given implementation on a given hardware is X time units, and the slowest execution time of the same is Y time units. The jitter of the system is then equal to $Y - X$.

Previously, Puschner [9, 10] has argued for WCET-oriented[2] programming, i.e. for algorithms in real-time systems to be optimized with regard to reduced jitter rather then reduced average execution time. According to Puschner, the main motivation for WCET-oriented programming is to make WCET-estimations more accurate, thus making scheduling easier and more efficient.

This is related to our work on logging structures presented in papers C and E that present algorithms with constant execution time. The motivations for a constant execution time differs from our motivation, Puschner argues that reduced jitter will: make control-algorithms function better, allow tighter scheduling, increase predictability and maintainability (as the number of conditional branches is reduced), and facilitate automated WCET-analysis.

1.2.3 Memory excluding checkpoints

A survey presented by Plank et al. [8] presents previous work on *memory excluding checkpoints*. The concept of memory excluding checkpoints is as follows: as the goal of checkpointing is recreation of a previous system state at a later point in time, a checkpointing algorithm is only required to log the data that cannot be deduced by other means.

Plank et al. state that there are two distinct approaches to exclude memory from checkpoints: To ommit dead memory, or to ommit read-only-memory. The goal of the first approach is to identify the memory that is no longer needed by the application (compare to garbage collection), and exclude this memory

[2] WCET is a common abbreviation for Worst Case Execution Time - in this context meaning the longest execution time possible

from the checkpoint. The goal of the second approach is to exclude the data which has not changed since some known system state (for example another checkpoint, or the initial state of the system). In our context, primarily read-only memory exclusion seems as an attractive option as it allows exclusion of memory that can be recreated by other means.

The challenge that we face is to minimize the size of checkpoints with out inferring a jitter into the system.

1.2.4 Garbage collection

There is a close familiarity between logging structures and garbage collection algorithms; both deal with releasing allocated records (or objects) in order to provide space for more recent data. Differences however, can be found in the criteria for collection/eviction; logging structures attempt to identify the “least useable” space, while garbage collection algorithms identify unused space.

1.2.5 Deterministic replay

Our deterministic replay technique, which supports timely replay of non-deterministic events such as interrupts, preemption of tasks, inputs from the environment, and distributed transactions, is presented in a number of publications [12, 13, 14, 16, Paper B, Paper F, Paper G]. The contributions of our technique include integration of the replay technology into Commercial-Of-The-Shelf (COTS) an Integrated Development Environment (IDE) (see Paper G), the use of memory excluding checkpoints as described by Plank et al. [8] (see Paper B), and the choice of real-time systems as target environment while using probes implemented in software (see Paper G). Further, as a validation of the replay technique in general and our instantiation of that technique in particular, the technique has been shown to work in a large state-of-practice real-time system that use a COTS operating system (see Paper G).

1.3 Published and preliminary results

During the work, some opportunities for publication have arisen. In this section, we survey the published and the to-be-published material that has been authored or co-authored by Joel Huselius. We differentiate between those publications that are included in the thesis, and those that have been left out. Generally, publications that have been left out have either seen only little input

from Joel Huselius, or present very early and unfinished work (such as Paper E).

1.3.1 Papers included in the thesis

Paper A

Joel Huselius. *Debugging Parallel Systems: A State of the Art Report.* MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE, September 2002, Mälardalen Real-Time Research Centre, Mälardalen University.

This paper surveys the field of debugging parallel and/or real-time systems with respect to both state-of-the-art and -practice. It includes a problem formulation, a listing of constraints on solutions to the problem, descriptions of previously published scientific solutions, descriptions of products available on the commercial market today, and a listing of the types of faults that may occur in computer systems.

Contribution from Joel Huselius: Mr Huselius is the sole author of the paper.

Paper B

Joel Huselius, Henrik Thane and Daniel Sundmark. *Starting Conditions for Post-Mortem Debugging using Deterministic Replay of Real-Time Systems.* In Proceedings of the 15^{th} Euromicro Conference on Real-Time Systems (ECRTS03), pages 177-184, Oporto, Portugal, 2^{nd} - 4^{th} of July 2003.

This paper discuss the issue of starting a replay execution based on a logging effort with memory excluding checkpoints. The technical contribution is a method for doing so, and a listing of the system requirements that must be fulfilled in order for it to work. The proposed method for starting replay has been integrated into a commercial-of-the-shelf development environment, and shown to work in an industrial starte-of-practice real-time system (Paper G).

Our method treats each task independently; as checkpoints are not coordinated between tasks, there is no globally consistent recovery line (see Paper A). Basicly, in order to start the replay, each task is first restarted with the same parameters as during the reference execution. After that a task has reached a pre-defined state (a local starting point), the state of the task is replaced with one from the log. As all tasks have reached their starting

points, the replay is commenced. Note that the only items that are required to be checkpointed are data variables whose values may have been dynamically altered during the execution. Thus, as some data may be ommitted, the checkpoints are memory excluding.

During the reference execution, checkpoints and other entries compete for space, it must therefore be ensured that local starting points are visited frequently enough during the reference execution so that checkpoints from them are still available in the log. This can lead to that the developer is forced to define multiple consecutive starting points in the same task.

One of the conclusions of the paper is that (because of multiple consecutive starting points), if no mechanism is deployed that can guarantee the availability of some required log-entries, it is not possible to guarantee the feasibility of the replay execution.

Contribution from Joel Huselius: Mr Huselius took the initiative to write the paper, he was the main author, and the coordinator of the effort. The technical contribution from Mr Huselius concerned the listing of system requirements posted by the contribution, the new terminology introduced,[3] and the work on multiple consecutive starting points.

Paper C

Joel Huselius and Henrik Thane. *Recording for Replay of Sporadic Real-Time Systems.* A version of this paper has been submitted for publication.

In Paper B, we concluded that multiple consecutive starting points may prevent deterministic replay when using memory excluding checkpoints. As a solution to this problem, Paper C presents the logging structure ECETES, which can be compared to a garbage collection algorithm for accumulated data. The logging structure is an extension to the CETES-algorithm presented in Paper E. The hypothesis of the paper is that ECETES, in sporadic systems and/or systems with multiple consecutive starting points, has a more efficient memory utilization then previous (FIFO) solutions.

The paper presents a comparison criteria for logging structures: The shortest interval of replay (SIR) is the period under which all tasks of the system are replayed, we note that it is only effectively possible to find and identify bugs executed during this interval. By comparing resulting SIR's for different logging structures on given system executions, it can thus be determined which

[3]Reference execution, potential-, global- and local- starting point, eviction scheduler, and multiple consecutive starting points.

of several that is the most appropriate logging structure for the given system.

Using the proposed method of comparison and the requirements found in Paper B, the paper presents an evaluation that support the proposed hypothesis. The evaluation is performed by means of simulation, and a set of three different simplistic system architectures are investigated. It is concluded that ECETES is the most suitable logging structure for sporadic real-time systems, and that ECETES should be the logging structure of choice for systems with multiple consecutive starting points.

Contribution from Joel Huselius: Mr Huselius took the initiative to write the paper, he was the main author, and the coordinator of the effort. The technical contribution from Mr Huselius concerned the motivation of the work, the implementation of the ECETES, LFIFO, and the simulator, the new terminology introduced,[4], the presentation of the simulation results, and the overhead messurements.

1.3.2 Published papers not included in the thesis

Paper D

Joel Huselius, Henrik Thane and Daniel Sundmark. *Availability Guarantee for Deterministic Replay Starting Points in Real-Time Systems.* In Proceedings of the 5^{th} International Workshop on Automated and Algorithmic Debugging (AADEBUG), Work in Progress Session, pages 261-264, Gent, Belgium, 8^{th} - 10^{th} of September 2003.

This paper discuss how algorithms such as ECETES can guarantee the possiblity to perform a correct replay execution. It is to be considered as an early paper on the same subject as Paper C.

Mr Huselius took the initiative to write the paper, he was the main author, and the coordinator of the effort. The technical contribution concerned the motivation of the work and design and implementation of the ECETES.

Paper E

Joel Huselius. *Logging without Compromising Testability.* MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-87/2002-1-SE, December 2002, Mälardalen Real-Time Research Centre, Mälardalen University.

[4]Shortest interval of replay, incubation period, logging structure, used starting point, and unneeded records.

This paper introduce a novel algorithm (the Constant Execution Time Eviction Scheduler, ECETES) for the activity of handling memory resources when logging data form a monitoring process. The contraints on the activity is discussed, and an algorithm that respect these constraints is presented. A major drawback of the presented solution is the limitation that all entries be the same size. In order to remedy this drawback, the work described in this paper evolved into the effort presented in papers C and D.

Contribution from Joel Huselius: Mr Huselius is the sole author of the paper.

Paper F

Henrik Thane, Daniel Sundmark, Joel Huselius and Anders Pettersson. *Replay Debugging of Real-Time Systems using Time Machines.* In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), pages 288-295, presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD03), Nice, France, 22^{nd} - 26^{th} of April 2003.

This paper describes the general techniques behind deterministic replay. It contains background information for Paper B and Paper G.

Contribution from Joel Huselius: Mr Huselius was engaged in the initial work, contributing with ideas during discussions. He was also involved in the writing process.

Paper G

Daniel Sundmark, Henrik Thane, Joel Huselius, Anders Pettersson, Roger Mellander, Ingemar Reiyer and Mattias Kallvi. *Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies.* In Proceedings of the 5^{th} International Workshop on Automated and Algorithmic Debugging (AADEBUG), pages 211-222, Gent, Belgium, 8^{th} - 10^{th} of September 2003. Also available as MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-96/2002-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University.

Experiences from two industrial case studies performed at ABB Robotics and SAAB Avionics are relayed in this paper. This work first identified the need for making use of memory excluding checkpoints during recording - a design choice that require the results presented in paper B.

Contribution from Joel Huselius: Mr Huselius was engaged in the initial work, contributing with ideas during discussions. He was also involved in the initial investigation of the ABB target system and the writing process.

Paper H

Joel Huselius. *Source-Code to the ECETES Logging Strategy.* Technical Report, Department of Computer Science and Engineering, Mälardalen University, August 2003.

The paper provides code to the ECETES implementation described in Paper C. Also other code used in the evaluation procedure of the ECETES is found here.

Contribution from Joel Huselius: Mr Huselius is the sole author of the paper.

1.4 Conclusions

In this thesis, we have been concerned with the process of preparing for replay of real-time systems. Paper A describes the state-of-the-art in the field of debugging. During the subsequent work with Paper F, we identified the need for a thorough description of a working method for how to start a replay-session from other than the initial state of the system, if memory excluding checkpoints (see Section 1.2.3) are used. In Paper B, we described such a method, and the system requirements that our method impose on the system. Among other things, that work revealed the need for dynamic logging structures that can guarantee the success of our method. We then presented such a logging structure in Paper C, by introducing ECETES. In that paper, we also describe a comparison criteria for logging structures, and present an evaluation that use that criteria to support our thesis that ECETES outperforms a FIFO solution in sporadic real-time systems.

In summary, this work has led to that the system requirements of replay are better known, and that the memory overhead of monitoring for replay is reduced.

1.5 Future work

The work presented here has left some leads to interesting work in the area of logging structures, the following are the leads that will be pursued in our future work.

The proposed logging structure ECETES will be further developed to utilize memory resources even better. Specifically, this includes pursuing the *unneeded*-marking of redundant records, and improving the selection technique. Also the validation process used in Paper C will be improved to give stronger evidence to our thesis. We plan to model a real-world system to perform the evaluation on, and to evaluate the unneeded-marking, which requires the verification to acknowledge that entries may individually have different mappings to records (i.e., 1-2, 1-4, and 1-6 -mappings in the same system). The work with finding other logging schemes, fundamentally different in their functionality then ECETES, will also continue.

Furthermore, issues remain in the larger field of debugging:

The taxanomy presented by Dionne et al. [2] will be extended to respect also temporal correctness and classes of bugs that may be found.

A hindrance to bringing replay technology into the idustry is the large perturbation caused by the recording effort on the target system, particularly checkpointing represent a large portion of this overhead. In our previous work (see papers B and G), we made use of a simple off-line memory exclusion technique to lighten the overhead of checkpointing; more work will be spent on developing new, or adapting old [8], techniques that can deliver under the posted requirements (the same requirements as those posted on logging structures).

Apart from using memory excluding checkpoints, the perturbation of logging can also be reduced by considering design and architecture decisions with respect to debugging. Knowledge about the replay technique and the memory exclusion algorithm can allow programmers to minimize the size of the state that must be logged. Knowledge about the way that the replay is initiated (Paper B) can be used to minimize the number of potential starting points, and therefore also minimize the overhead caused by using many queues (see Paper C). During our continued work, we will collect principles, advice, and guidelines for system design that will lead to a reduced overhead of recording.

Bibliography

[1] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A pertrubation-free replay platform for cross-optimized multithreaded applications. In *In Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, April 2001.

[2] C. Dionne, M. Feeley, and J. Desbiens. 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.

[3] P. Dodd and C. Ravishankar. Monitoring and debugging distributed real-time programs. *Software - Practice and Experience*, 22(10):863–877, October 1992.

[4] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

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

[6] J. Mellor-Crummey and T. 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.

[7] R. Netzer and J. Xu. Adaptive message logging for incremental program replay. *Parallel & Distributed Technology*, 1(4):32–39, November 1993.

[8] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, 1999.

[9] P. Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proceedings of the* 8^{th} *IEEE International Workshop on Obj ect-Oriented Real-Time Dependable Systems*, January 2003.

[10] P. Puschner. Hard Real-Time Programming is Different. In *Proceedings of the* 11^{th} *International Workshop on Parallel and Distributed Real-Time Systems*, April 2003.

[11] K.-C. Tai, R. Carver, and E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):280–287, Januari 1991.

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

[13] H. Thane. Time machines and black box recorders for embedded systems software. *European Research Consortium for Informatics and Mathematics News*, (52):32–33, January 2003.

[14] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265–272. IEEE Computer Society, June 2000.

[15] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[16] H. Thane and D. Sundmark. Debugging using time machines: replay your embedded system's history. In *Proceedings of the Real-Time & Embedded Computing Conference*, November 2001.

[17] J. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[18] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the* 30^{th} *Annual International Symposium on Computer Architecture*, pages 122–133, June 2003.

[19] F. Zambonelli and R. 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.

Chapter 2

Paper A: Debugging Parallel Systems: A State of the Art Report

Joel Huselius
MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, September 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- and multi-processor as well as 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 during debugging than less complex systems. 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.

2.1 Introduction

To debug sequential software can be considered as fairly straightforward - as long as each execution is deterministic and repeatable, the cause of an observed failure can always be found by existing methods.

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 [60]. Some inputs to the system may not be controllable, or even visible, resulting in seemingly non-deterministic behaviour that cannot be repeated at command. Further, assuming traditional techniques used for debugging sequential systems, we can see that they are insufficient for the task of debugging; even if we sense the presence of a bug by observing a failure of the system, it is not always given that we can derive the location of the bug that caused the failure. Information about the location of a bug is required in order to repair the system successfully.

One of the intentions of this report is to investigate the issues that make debugging parallel systems so hard. We will describe the generally accepted method for debugging parallel systems through the use of execution *recording* and execution *replay*, and describe the topics that are important to cover in implementations of the two. In the context of recording, we discuss issues such as:

1. monitoring, that describes the act of observing the system during runtime
2. logging output from the process of monitoring
3. the *probe effect* [9] (see Section 2.4.1) that can lead to unintended system behavior,
4. the correlation problem (see Section 2.4.1) that must be solved when recording the execution of distributed systems,
5. and the observability problem 2.4.1 that states that the system must be open enough to allow monitoring.

In the context of replay the problems, we discuss issues such as:

1. the *completeness problem* (see Section 2.5.2)
2. and the *bystander effect* (see Section 2.5.1).

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. As the industry (understandably enough) is reluctant to make detailed information about their products publicly available, the focus here will be on state of the art.

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

2.1.1 Outline

The outline of this report is as follows: Chapter 2.2 provides explanations and definitions for some fundamental terminology relevant to the rest of the report. Chapter 2.3 describes the categories of potential errors which may occur in parallel systems.

Thereafter, the two chapters that follow discuss the two sub-activities in the widely accepted basic method for debugging parallel systems: Chapter 2.4 discusses how recording can be performed, the problems that are encountered, and discusses different approaches and tools found in the literature. Chapter 2.5 discuss issues that must be respected when constructing a replay-mechanism, some different approaches found in the literature are also discussed.

Chapter 2.6 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 2.7.

2.2 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 the book “Distributed Real-Time Systems: Monitoring Visualization and Debugging and Analysis” by Tsai et al. [64, 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.

2.2.1 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 source code, but more complex constraints.

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 may be phase-shifted to each other.

Tasks have a *periodicity* at which it emitts *jobs*. A job is the execution of one iteration of the code of the task. The task constraints are modelled as *deadlines*, *release times*, *jitter*, etc. In order to model phase-shifting between tasks, a real-time task has an *offset* that specifies the earliest point in time from its release time that a job of 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, its contribution to the computation cannot be considered usable. The severity of such a failure may be grave (for hard[1] real-time systems), but there are systems (soft[2] real-time systems) which are designed to allow some amount of deadline-misses.

For example, the human task of sleeping should emitt a new job each night. Hence, the periodicity of sleeping is approximetly 24 hours, the execution time of sleeping is perhaps averaging on eight hours, even if the worst-case execution time may be much longer. An alarmclock can be set to indicate the deadline of the task, a deadline miss would be oversleeping. Further, many people have arrival-jitter in their sleeping task as they do not fall asleep at the same time every night. The release time of the task could describe the activity of trying to fall asleep. Waking up in the middle of the night, having to go to the privy, would be a context-switch to another task. This multi-tasking will obviously increase overhead - it takes time to fall asleep again once you have awoken.

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

[1] Hard real-time systems: Rockets, airplanes, etc.
[2] Soft real-time systems: Multimedia streaming, toys, etc.

socks have precedence over shoes.

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 jobs.

2.2.2 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 [57, 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 system state, that is an error. The error leads to a failure of the component operation if its presence is visible or sensed outside the component.

There are two ways of avoiding faults in a system [42, 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.

2.2.3 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 exist in parallel systems, but we will in this section make a small example. In his Ph.D. thesis [57, s3.2.1.3;p27], Thane recalls 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. They 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 itself (or querying multiple nodes) as the answer lies within the scope of the potentially valid answers - it is potentially true that fish is on the menu. 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.

2.2.4 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.

2.2.5 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 where a resource (for example a CPU) is more or less transparently shared in time between two or more tasks.

Hardware

The computing environment may be a distributed system, or a multiprocessor architecture, complex hardware can be heterogeneous or homogeneous, i.e. the nodes of the system are not necessarily uniform with respect to their hardware architecture: instruction sets, operating systems, computation capabilities, and external resources may differ between nodes. As different nodes in a distributed system have individually differing temporal propagations, timing is an interesting factor that may complicate 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 allow the construction of System-on-Chip (SoC) hardware. SoC-technology allows designers to place an entire system on one silicon dye, as this allows (among other things) reduced contact with the slower off-chip components, performance is increased. However, the reduction of off-chip information flows limits our visibility of the system - many of the hardware transactions between on-chip components may seem invisible and uncontrollable for external hardware [16].

In order to create a greater understanding for our term “complex hardware”, we will 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.

There are three different formats of floating-point representations 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, which 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 instruction 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.

According to Weaver and Germand [68], 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.

There are quite a lot of possible traps that may occur. In a file that has a (version dependent) path similar to the string `/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 that will slow down the execution of the trap handler.

Because of the nature of the invocation of traps, SPARC differentiates between four different categories of traps [68]; Precise, Deferred, Disrupting, and Reset traps. An instance of a trap belongs to one of these four categories.

Precise Traps Precise traps are results of the execution of a special instruction whose objective is merely 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 were 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 allow the program state to be changed between the dispatching and the execution of the instruction (see Section 2.2.5 for an example).

If a deferred trap and a precise trap occur 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.

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 if 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. Thus, the fault hypothesis (see Section 2.2.3) will direct which divergencies that can be allowed.

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 commercial-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 modelled 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 objective is to remedy both these issues.

In this report we explain general problems in debugging software, and also explain which other problems that 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.

2.2.6 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.

Cyclic debugging

The normal way of debugging software systems is to repeatedly use for example a debugger that has facilities like stepping, breakpointing, 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/her search for the suspected error. This process is normally referred to as *Cyclic(al) Debugging* [22, 30], 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 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, if the breakpoint had not been present.

Recording 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 *recording* (subactivities *monitoring* and *logging*) one execution of the system that is to be debugged to such a level of detail that we can then replay 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 and logging systems will be discussed in detail in section 2.4, 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 recording as the *reference execution*. In section

2.5 we survey different methods for, by using recording output, replaying an instance of a system. This facsimile of the replay execution is callled the *replay execution*.

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

2.3 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 that may occur in sequential software [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, 37]. Thane [57] 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.

2.3.1 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 known phenomenon's, but we provide a short description here.

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, unknowingly, be 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.

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 2.1): 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

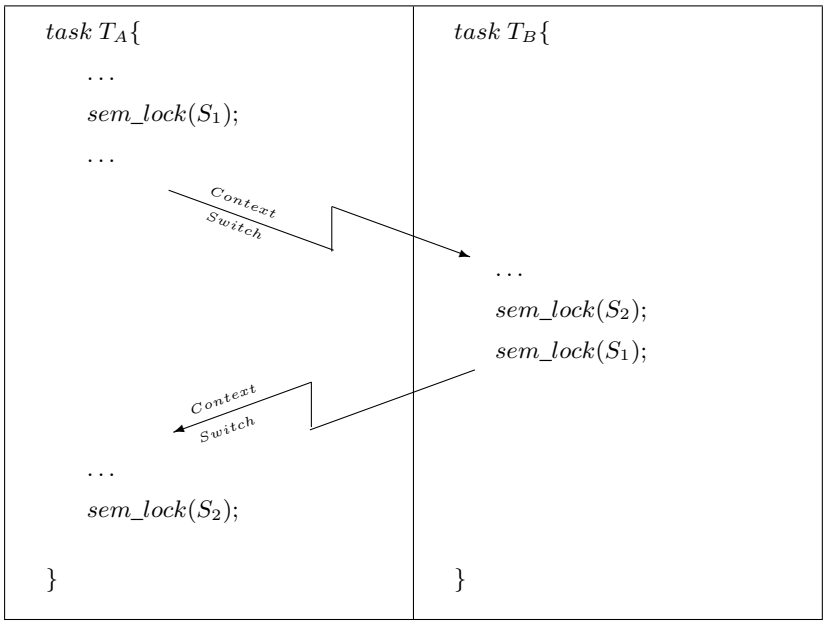

Figure 2.1: Example of a deadlock.

already owns that semaphore.

Since neither T_A nor T_B can continue their 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 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.

Livelock

Under livelock, in difference to deadlock, a system is locked in an unintended loop of instructions that do not allow further computations on the intended task; when the overhead approaches 100% of the utilization. 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 2.3.3).

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 the behavior of the protocol when the network experiences 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[3] 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]The author, who is of Swedish origin, notes that this classic child's play is called "Sten Sax Påse" in Swedish.

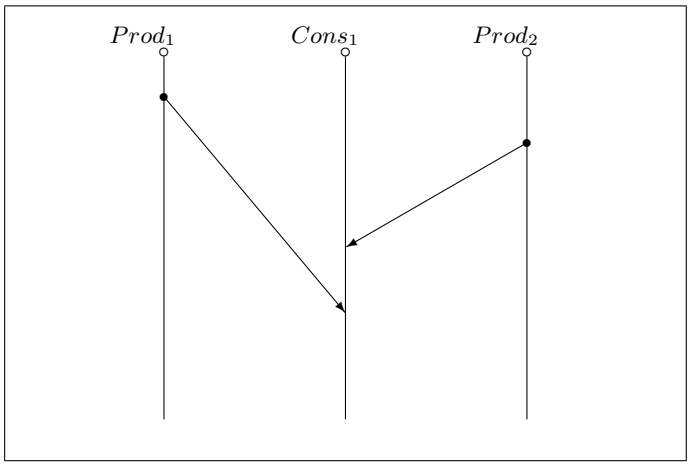

Figure 2.2: Example of a message race.

2.3.2 Race conditions

The popular description of a race condition is as follows: when two or more system entities[4] 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. For example, a race can be found in network communication, see Figure 2.2. 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 [37], see also Netzer [34], where they search for race conditions in *prefixes* [34, s3.3.1;p21] of a particular

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

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 2.3 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 2.3, which exhibits the *actual race* (see below), we can also find other execution orderings that are prefixes of the original execution. Part (2.) in Figure 2.3 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*, or *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 2.3 (1.)[5] or (2.)). Actual data races exists if and only if there exists at least one data race in the original execution (in Figure 2.3 (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 2.3 (4.)).

In equivalence, 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

[5] In the case of part (1.) in Figure 2.3, note that an execution can have the same sequence of events - without being equivalent to the original execution in all other aspects.

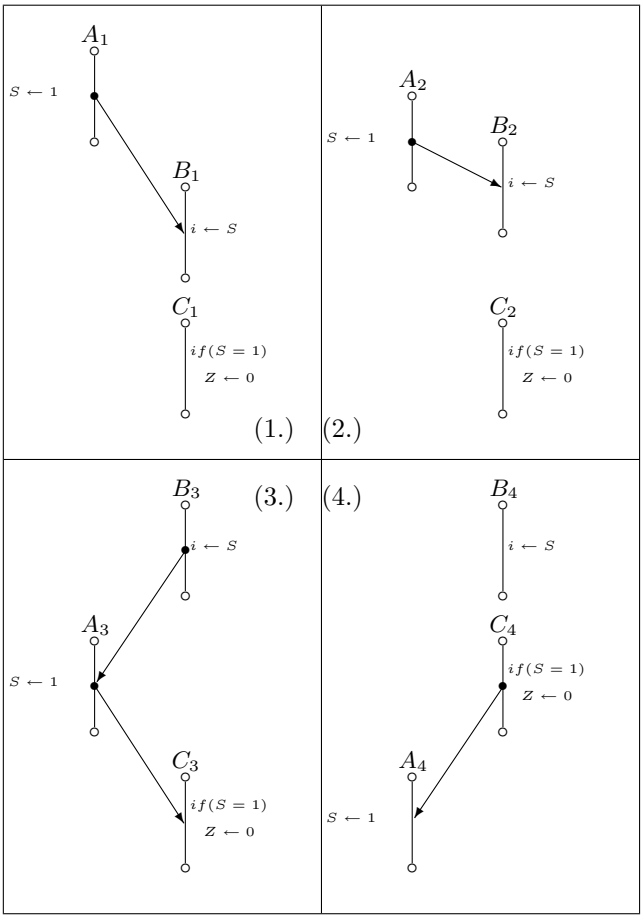

Figure 2.3: Data races; (1.) actual execution. (2.) feasible execution. (3.) feasible execution. (4.) infeasible execution.

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 [43, 45] which uses the ROLT method described in Section 2.4.5. RECPLAY can detect unwanted race conditions during the replay 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 recorded 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.

2.3.3 Real-time errors

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

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.

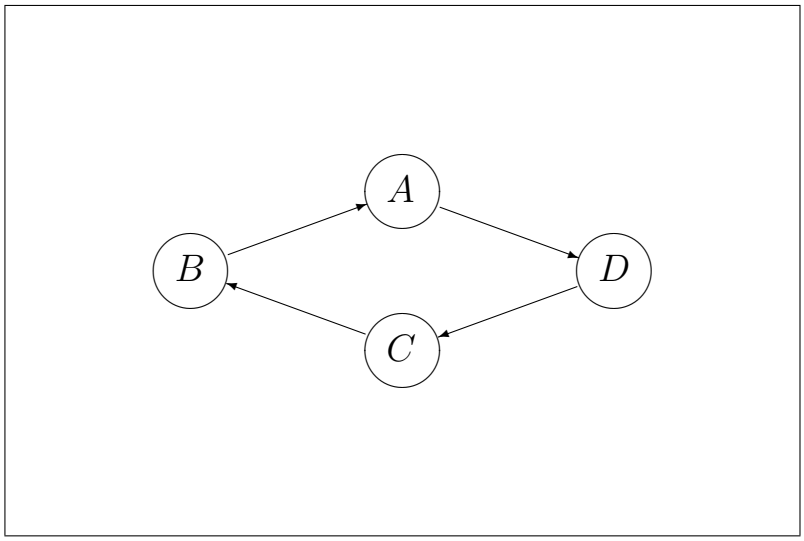

Figure 2.4: A precedence graph.

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 2.4.

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. [64, s9.1.1;p192] provide a classification of causes of timing errors:

Computation Causes If a greedy task requires more resources than has been granted, other tasks may find themselves with too few resources to complete their task. For example, this problem can easily arise if the measured Worst Case Execution Time (WCET) is lower than the real WCET.[6] Best results are achieved by estimating a WCET which is as tight as possible, but never underestimates the actual WCET. 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 2.3.

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 unschedulable. 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.

[6] Note that the measured WCET cannot exceed the actual WCET, wherefore a safety margin is often added to the measured value.

Synchronization Causes We have discussed the occurrence of synchronization problems in previous sections (see for example deadlocks and livelocks in Section 2.3.1), and they may of course also 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.

2.4 Recording, monitoring and logging, execution traces

Recording is an activity with two subactivities, *monitoring* and *logging*. Monitoring, according to McDowell and Helmbold [30], is the process of gathering information about a program's execution into *entries*. Further, logging is defined as the process of storing *records* of the entries to a medium. By monitoring the execution of a program, and then logging the entries, we can analyze the execution off-line in some form of model of the platform that was used, an issue which will be covered in Section 2.5 - the current chapter will deal with the problem of performing a recording. Implementations ranges from additional software that is added to the system at some level, to tailored hardware, to hybrid approaches.

Because recording provides us with detailed information about a systems execution, detailed enough to recreate the execution, we can apply cyclic debugging to a recorded 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 2.2.6.

2.4.1 The probe effect and the correlation problem

There are similarities between the probe effect and the correlation (or observer-) problem. In this section we explain the two, and point out differences and similarities. We shall also discuss the observability problem.

The probe effect

The *probe effect* [9], which is another name for *Heisenbergs uncertainty principle*[7] when applied to software engineering [24, 30, 50], can become

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

visible when code is added or removed to a system, when breakpoints are used to debug the system, or when the system is modified in some other way that will effect 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. This would have lead to a failure, and a debugging-effort is launched.

In order to debug the system, we would like to probe into the state of the tasks so that we could determine the cause of the problem. However, if we implement 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 will 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 2.5.3).

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, 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 [57, s4.3.3;p42]. Thane starts with the same premise that Schütz did; 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. He then states that this requirement is satisfied if the task from where the code is removed has the lowest of priorities among the tasks in the system (appart from the idle task) and it is 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.

Note that these solutions are only feasible under the assumption that the operation of the hardware (instruction pipelines, caches etc.) is not affected by the removal of the probes.

The correlation problem

In "Fundamentals of Distributed System Observation" published 1996 [8], Fidge describe 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. We shall refer to this as the *correlation problem*.

According to Fidge, we may divide the correlation 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 the 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 [42]. 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.

The observability problem

It should be noted that Schütz discusses a subject which he calls observability [48]. Schütz states that a system must be *observable*, meaning that it must be

possible to extract sufficient information from the system. What is "sufficient" is determined by the present fault hypothesis.

Conclusion

Thus, we may conclude that the probe effect causes changes to the program execution, whereas the correlation problem affects our perceived view of the program execution, and the observability 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 correlation problem.

2.4.2 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 much execution resources a task has consumed between two records 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 required for the execution to reach the instruction from a known starting point 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.

Consistent temporal view

An issue that may arise 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. If this is a requirement of the debugging method, some measure must be taken. Tightly coupled parallel systems, and multitasking single-node systems, are able to produce a correct ordering because all system entities depend on the same real-time clock [64, s3.1;p51]. But, because of the correlation problem (See Section 2.4.1), 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* [64, 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].

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” written in 1987 by Kopetz and Ochsenreiter [17]. The paper presents an algorithm for global clock synchronization.

2.4.3 Global state

One problem that one has to face when implementing a strategy that use 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 dependency-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.

Note that, assuming that checkpoints are used, only making one checkpoint in the beginning of the execution is not sufficient. Because very long recording sessions require very much memory resources in order to keep the logs, and those resources are finite, it is required that old log records be evicted as the memory is exhausted. The eviction is made in favor for newer records, that are of 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 records [53]. 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 2.4.5).

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 records, wherefore other structures could be preferable (see Section 2.6.5).

Checkpointing

The reason for making checkpoints of a system is to be able to start over with the execution at some later point [38, 69]. 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 record - 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 [69] 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 [62]. 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.

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" [67]:

> "...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].

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 [67], there are mainly three different strategies to distributed checkpointing: Uncoordinated checkpointing, coordinated checkpointing, and log-based techniques. Chow and Johnson [4] 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 violates 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 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.

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” [41] 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 [57, 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 of the sequences of instructions executed by the processors(s), and relevant[8] timing information regarding that execution. The data flow of the system is represented both by the relevant[9] alterations of system data during run-time, and timing information regarding these alterations. In order to successfully replay the recorded system, both the control- and the data-flow must be monitored with a sufficient degree of detail which is defined by the replay technique used.

2.4.4 Sufficient 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[10] 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 about 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 reproduced, but every abstraction opens the door for errors to escape the debugging process unnoticed. Thus, we must have a fault hypothesis (see Section 2.2.3) before we can define the monitoring activities in the system.

Logging

The product of a monitoring activity can be logged on to a 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 recorded execution of the system.

[8] 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.

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

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

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 2.2.2) 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 input to the process of calculating 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.

2.4.5 Discussing recording approaches

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

It is also possible to classify recording 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.

Hardware

Hardware recording 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 [64, s2.3;p37].

Basic approaches to hardware recording 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 logged. Another problem with hardware implementations is that they must look at very low level information [64, 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 recording 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 recording 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 [64, s5;p103]. SoC solutions are not observable as they limit the insight to the internals of the system, and it is therefore more difficult to construct hardware implementations for these systems provided that they are not incorporated on the chip [16]. A solution could be to move also the recording into the chip, but this is approach is of course only available to the designers of the device. Monitoring primitives implemented in hardware cannot be added to a SoC-design at a later stage. Thus, SoC technology is obstructing the use of commercial-off-the-shelf components where recording is required. We shall, in Section 2.4.5, survey a proposed hybrid methodology for recording the executions of SoC's.

In their work on a "non-interference monitoring and replay mechanism", Tsai et al. [65, 66] present a hardware solution for monitoring by bus snooping. In their solution, they use a duplicate processor that executes in parallel with the target. As the thought recovery line (Section 2.4.3) is reached, the duplicate processor is frozen and its state is logged - that state can then be used during replay to start the replay from. Even though they claim in the title of their papers on the subject that their method provides these services without interference of the target environment, they do admit that they require the use of one occurrence of an interrupt to synchronize the two processors at the start of the monitoring session (which is not necessarily identical to the start of the system).

Boundary Scan IEEE Standard 1149.1 defines test logic [13]. The standard is a result from work by the Joint Test Action Group (JTAG)[11] 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 replay of a execution through the use of that interface.

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

The replay 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 replay 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 with a large shift register, a solution that infers large temporal penalties on the replay of the execution. In the case of recording, it seems that the same problem provides a limit for the granularity of the 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 discusses 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 recording the executions of complex processor architectures, especially those with on-chip caches, as it uses recording 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 memory resources can be explained by the fact that real-time monitoring requires the recording mechanism to be prioritized over the application, thus leading to intrusive recording.

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 necessarily intrusive on the target functionality, not even 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 are 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 provides a set of instructions that 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 replay 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 replay 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.) that 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 completely hardware-controlled facility that allows monitoring of program flow while the processor runs in real-time. 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 that are also used to describe 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.

Software

Similarly to the cyclic debugging approach described above, software implemented recording mechanisms 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 [64, 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 [57]. The black-box may then be used if a released system experiences 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 logging of the monitoring data may present a problem.

Software monitoring can either be performed at system, or process (task) level [64, 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 [57, 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 monitor 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.

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 replay of the system execution.

Zambonelli and Netzer et al. discusses 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” [38, 69]. The authors state that logging all messages is resource demanding during the reference execution, but recreating all messages during the replay 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 replay it is theoretically possible to recreate all messages that occur in the system by replaying 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 replay. 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 [59] has provided this feature (see Section 2.5.4).

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 vertices, 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.

In “A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks” published in 1994 [26], Levrouw et al. presented ROLT, the Reconstruction Of Lamport Timestamps. The method is an improvement of Netzers method described above. Instead of using vector clocks, as Netzer, the authors use Lamport clocks (see Section 2.4.2). 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 log an entry only in the nondeterministic case. The penalty inferred by the use of this optimization is that a loged entry (a record) 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)[12] programs. The debugger uses a reocord/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.

Hybrid

According to Tsai et al. [64, s5.1;p104] hybrid implementations 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.

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

From Applied Microsystems (www.amc.com) comes the CodeTEST Trace Analysis tool that provide 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 source 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 can be left in the system in order to avoid probe effect related problems.

In “A Hardware and Software Monitor for High-Level System-on-Chip Verification” [49], El Shobaki and Lindh presents a method for recording the execution of SoC’s 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 2.4.5), 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. However, integration of the MAMon into a SoC-system is only available to the hardware-designer.

Discussion

It seems that different monitoring implementatations monitors the system at different levels; hardware implementations monitor low-level details, while software probes monitor the high-level flow of the implementation. Consequently, 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 certain amount of the water in a cistern. To monitor the system from a level above the software could then be to monitor the amount of water in the cistern, in order to evaluate the implementation of the control algorithm.

We note that there are different levels of monitoring, and that each level have different advantages and drawbacks. Thus, we can state that monitoring at different levels is not strictly comparable. It is therefore likely that several differenet levels of monitoring should be used concurrently, 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 2.2.4), or if parts of the log has been forfeited (see Section 2.4.3), 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.

2.5 Replaying the execution of a computer system

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

2.5.1 The stampede effect and the bystander effect

There are similarities, not only between deadlock and livelock, or between the probe effect, the correlation problem, and the observability 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]:

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, off-line-assumptions about

the state of shared resources 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 a second task that is still alive replaces the erroneous data with correct values before the whole system terminates. It is then impossible to, by only viewing the memory state of the crashed system, determine what went wrong.

The bystander effect

Similairly to the stampede 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.

The bystander error is similar to the input error described above, where a task erroneously makes use of faulty input received from another task, but the bystander error concerns input through interference that the affected task is unaware of. For example if the data used by a task T_A is modified by another task T_B without the knowledge of T_A. 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 input.

Conclusion

When constructing a model for replay 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 become visible if the system is allowed to continue execution past a failure of a task without reporting this to the user. If the replay was not a success in terms of sensing an occurred failure, one of two problems may follow: Another bystander task may 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 replay mechanism has no clear sense of the system specification, but is also a potential problem during the activity of recording the execution, failure to log a change to a monitored entity may produce the same problems.

2.5.2 Irreproducibility and completeness

The irreproducibility effect and the completeness problem are similar to each other in that both of them only emerge in nondeterministic[13] parallel programs [50] [18, s4.2.3;p87].

Irreproducibility effect

The reproducibility problem, also known as the irreproducibility effect or the non-repeatibility 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 2.5.3 on regression testing), and also to distinguish between different bugs [18, s4.2.6;p94].

In his thesis, starting from a definition of deterministic systems compatible with Kranzlmüller's definition of non-deterministic systems (see Section 2.2.4), and a definition for *Partial Determinism*, Thane [57, 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.

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 solve this problem it is imperative that the nondeterministic elements of the system are recorded, an issue which we discuss in Chapter 2.4.

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

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

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[14] 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 to the system.

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 [58, 60, 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 monitored 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, 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

[14] 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 $(2^{64} \cdot 10^{-9})/(60 \cdot 60 \cdot 24)$, or approximately 200'000, days of testing.

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.

2.5.3 Regression testing

As a bug is identified, and an attempt to remove it has been made, two things must be confirmed: The fix must not have introduced further bugs in the system and 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 [59] 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 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 solve 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.

2.5.4 Uses of logs

McDowell and Helmbold [30] stated three 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.

Browsing

Browsing is the act of 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 the contents of the log.

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.[15] 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

[15] 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 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 replay executions are identical, this will guarantee that the two executions are identical.

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 using Software Instruction Counters (see Section 2.4.2), and also counting the function entry- and exit- points (there may be several different exit points from a function), executions logs are created. The logs will contain sufficient information to facilitate execution replay, 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 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.

Thoai et al. [63] present Shortcut Replay for distributed systems, which allows the replay to start from a state other then the initial state of the system. They use uncoordinated checkpointing (see Section 2.4.3), but show that replay does not have to consider the risk that orphan messages (see Section 2.4.3) complicates the process of finding a recovery line. In their technology, they do not make use of memory excluding checkpoints or other means of reducing the perturbation of recording.

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

recording 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].

Paik et al. [39] present their Concurrent Program Debugging Environment (CPDE), which provide replay for UNIX processes. By providing compiler options, they facilitate the creation of one executable version of the system that produces a log, and one that can be replayed by consuming the afore mentioned log. In their work, no reference to the way they correlate events and location in the code with unique markers (e.g., software instruction counters, or timestamps). Hence, they cannot replay asynchronous events such as interrupts etc.

In their method for performing replay of Ada programs, Tai et al. [56] assumes that the run-time system does not experience non-determinism due to interrupts, abort statements, dynamic task allocation, etc. Neither do they make any explicit refrences to how this can be solved. By transforming program P into two other programs, P' and P'', such that P' will produce a log that is consumed by P'' in order to force a deterministic execution, they can debug program P'. However, using this method will tempt developers to release version P of the program; due to the probe effect, this version is not verified.

In "Using Deterministic Replay for Debugging of Distributed Real-Time Systems" [59] Thane and Hansson describes a 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 implementation, 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.

Simulation

By using a simulator of the target system, and forcing it to behave in a way that will produce a 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 2.5.3), 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.

2.5.5 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 do 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 [40].

2.6 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.

2.6.1 Deterministic replay

We will in our future work concentrate on the simulated deterministic replay approach, using software implementations, 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 2.5.4) as method of replay would perhaps not suffer as much from this approach as replay and simulation (see Sections 2.5.4 and 2.5.4). 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 enforce 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 2.4.3 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 2.4.5, we imply that we may view such systems as nondeterministic or partially deterministic (see Section 2.5.2). The loss of some of the information that defines the execution may satisfy the rules for nondeterministic systems (see Section 2.2.4) if there is not a sufficiently large amount of task level (see Section 2.4.5) traces, in which case it may satisfy the criteria for a partially deterministic system.

Whether they are reproducible, or partially reproducible (see Section 2.2.4) remains to be seen.

2.6.2 Debugging component based systems

In Section 2.4.5, 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 2.4.3.

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.

2.6.3 Design patterns for design of observable systems

As we have pointed out in this report, an inherent problem with recording 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 recording mechanism.

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

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 2.4.5 and [38, 69] on Incremental Replay). Between each such task instance, at least all data which cannot be reproduced must be recorded. If all this data is stored in a easy to reach structure, this would certainly ease the recording 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.

2.6.4 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 seen which tool is the most efficient in some relevant aspect.

The reason for this insufficiency is that 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 2.4.5 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.

2.6.5 Efficient memory usage when logging

Stewart and Gentleman [53] mentioned the applicability of circular queues as an infrastructure when logging monitoring entries into records. It seems that the potential for keeping redundant information in such a scheme is larger then needed. This was also implied by Ronsse et al. [44], they stated that records should be evicted as soon as they are without use.

When using a circular queue structure, garbage collection is trivial; entries can be logged in chronological order on the medium, and as space is exhausted the oldest record is replaced with the newest one. 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 logged information. That is to say how many of the logged records that can be used in a replay. In the circular queue solution, no respect is paid to the relative context of the information that is expunged and the information that 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 logged 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 2.4.1, 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 logging of monitoring activities. The intention of that work should be to reduce the amount of unusable information that will accumulate in the log.

2.6.6 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 journals, conferences, and workshops are the IEEE Transactions Parallel and Distributed Systems, the IEEE Symposium on Reliable Distributed Systems, the ACM International Symposium on Software Testing and Analysis, the Workshop on Automated and Algorithmic Debugging, the Euromicro Conference on Real-Time Systems, and the IEEE Real-Time and Embedded Technology and Applications Symposium.

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. At www.elis.rug.ac.be/ELISgroups/paris/index.html, the PARIS group is presented. On the related site sunmp.elis.rug.ac.be/recplay/, they describe their project called RecPlay.

2.7 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 recording 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.[16]

Among recent dissertations in the field, we mention Henrik Thane [57] (2000), and Dieter Kranzlmuller [18] (2000).

[16] 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] A. Arora, S. Kulkarni, and M. Demirbas. Resettable vector clocks. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 269–278, July 2000.

[2] B. 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] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, March 1991.

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

[5] S. Clarke and J. McDermid. Software fault trees and weakest preconditions: A comparison and analysis. *Software Engineering Journal*, 8(4):225–236, July 1993.

[6] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[7] C. Dionne, M. Feeley, and J. Desbiens. 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] C. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.

[9] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

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

[11] S. Howard. A background 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] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[15] R. Kilgore and C. 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] H. Koehnemann and T. Lindquist. Towards target-level testing and debugging tools for embedded software. In *Conference Proceedings on TRI-Ada*, pages 288–298. ACM, September 1993.

[17] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *Transactions on Computers*, 36(8):933–940, August 1987.

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

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

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

[22] T. LeBlanc. Parallel program debugging. In *Proceedings of the 13th Annual Computer Software and Applications Conference COMPSAC'89*, pages 65–66, September 1989.

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

[24] C. LeDoux and S. Parker. Saving traces for ada debugging. In *Proceedings of the Ada International Conference on Ada in Use*, pages 97–108. ACM, May 1985.

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

[26] L. Levrouw, K. Audenaert, and J. V. Campenhout. 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] L. Lindh, J. Stärner, J. Furunäs, J. Adomat, and M. E. Shobaki. Hardware accelerator for single and multiprocessor real-time operating systems. In *the Seventh Swedish Workshop on Computer Systems Architecture*, June 1998.

[28] J. Lumpp, K. Sivakumar, C. Diaz, and J. Griffioen. 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, January 1998.

[29] C. MacNamee and D. Heffernan. Emerging on-chip debugging techniques for real-time embedded systems. *Computing & Control Engineering Journal*, 11(6):295–303, December 2000.

[30] C. McDowell and D. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[31] J. Mellor-Crummey and T. 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] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[33] D. Neri, L. Pautet, and S. Tardieu. Debugging distributed applications with replay capabilities. In *Proceedings of the 1997 conference on TRI-Ada*, pages 189–195, November 1997.

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

[35] R. 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] R. 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] R. Netzer and B. Miller. What are race conditions? - some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[38] R. Netzer, S. Subramanian, and J. Xu. 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.

[39] E. H. Paik, Y. S. Chung, B.-S. Lee, and C.-W. Yoo. A concurrent program debugging environment using real-time replay. In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 460–465, December 1997.

[40] C. Pancake and S. Utter. Models for visualization in parallel debuggers. In *Proceedings of the 1989 Conference on Supercomputing*, pages 627–636. ACM, November 1989.

[41] B. Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

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

[43] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *Transactions on Computer Systems*, 17(2):133–152, May 1999.

[44] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay and debugging. In *Proceedings of the* 4^{th} *International Workshop on Automated Debugging*, pages 5–18, August 2000.

[45] M. Ronsse, M. Christiaens, and K. D. Bosschere. Cyclic debugging using execution replay. In *International Conference on Computational Science*, volume 2074 of *LNCS*, pages 851–860, May 2001.

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

[47] M. Russinovich and B. 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] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[49] M. E. Shobaki and L. 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] D. Snelling and G.-R. Hoffmann. A comparative study of libraries for parallel processing. *Parallel Computing*, 8(1-3):255–266, 1988.

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

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

[53] D. Stewart and M. 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] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[55] F. Suárez, D. F. García, and J. García. 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] K.-C. Tai, R. Carver, and E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):280–287, Januari 1991.

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

[58] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of the 20th Real-Time System Symposium*, pages 360–369. IEEE, December 1999.

[59] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265–272. IEEE Computer Society, June 2000.

[60] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[61] H. Thane, A. Pettersson, and H. Hansson. Integration testing of fixed priority scheduled real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, December 2001.

[62] H. Thane, A. Pettersson, and D. Sundmark. The asterix real-time kernel. In *Proceedings of the 13th Euromicro International Conference On Real-Time Systems*, June 2001.

[63] N. Thoai, D. Kranzlmüller, and J. Volkert. Shortcut replay: A replay technique for debugging long-running parallel programs. *Lecture Notes in Computer Science*, 2550:34–46, January 2002.

[64] J. Tsai, Y. Bi, S. Yang, and R. Smith. *Distributed Real-Time Systems: Monitoring Visualization and Debugging and Analysis*. Wiley-Interscience, 1996.

[65] J. Tsai, K.-Y. Fang, and H.-Y. Chen. A replay mechanism for non-interference real-time software testing and debugging. In *Proceedings of the Conference on Software Maintenance*, pages 209–218, October 1989.

[66] J. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[67] Y.-M. Wang and K. 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.

[68] D. L. Weaver and T. Germand, editors. *The SPARC Architecture Manual*. PTR Prentice-Hall, 1994.

[69] F. Zambonelli and R. 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.

Chapter 3

Paper B: Starting Conditions for Post-Mortem Debugging using Deterministic Replay of Real-Time Systems

Joel Huselius and Daniel Sundmark and Henrik Thane. In Proceedings of the 15^{th} Euromicro Conference on Real-Time Systems, pages 177-184, Oporto, Portugal, July 2003.

Abstract

Repeatable executions are required in order to successfully debug a computer system. However, for real-time systems, interactions with the environment and race conditions in the execution of multitasking real-time systems software make reproducible behavior difficult to achieve. Earlier work on debugging of real-time software has established the use of a *deterministic replay*, a record/replay solution, as a viable approach to reproduce executions.

When combining the deterministic replay approach with infinite loop recorders (similar to black-box recorders in airplanes) for post-mortem debugging, it is essential that the recordings are sufficiently long and detailed in order to be able to re-execute the system. Basic problems however, are how to find a well-defined starting point within the recording, and how to find a reachable state in the rebooted/restarted system to match that instance? Previous work has not presented solutions to these fundamental problems, in this paper we do. We also present some implementation details from an industrial case study.

3.1 Introduction

Traditionally, debugging is performed by means of *cyclic debugging* [4]. After that a system has failed during test or operation, repeated re-executions of the system together with diagnostics is used to track down the suspected perpetrator; the bug. Typically, the diagnostic process is simplified by basic mechanisms in the debugging environment, e.g. interactive breakpoints, tracing, etc. Cyclic debugging of multi-tasking real-time software is distinguished from cyclic debugging of single tasking non-real time software by the need to account for race conditions and potential non-deterministic re-executions, as well as non-deterministic inputs [14]. Earlier work on debugging of real-time software has established the use of *deterministic replay*, a record/replay solution, as a viable solution to create repeatable executions [1, 6, 13, 17, 15, 19]. When combining the deterministic replay approach with infinite loop recorders (analogous to black-box recorders in airplanes) for post-mortem debugging of embedded systems, it is essential that the recordings are sufficiently long and detailed in order facilitate re-execution of the system. However, while saving sufficient amounts of information, the limited amount of resources (temporally and spatially) available must still be respected. Fundamental problems to solve are:

- how to find a well defined starting point in the recording that matches the state of the restarted real-time system, and
- how to find/change the startup state of the system to match one instance within the recording?

In this paper we explain, and present solutions to the two problems described above, which in previous related works have not been addressed.

We will also present details from a recent case-study performed on an industrial robot system that is using the VxWorks operating system. The case-study was a feasibility test of the monitoring/replay methodology in general, and our method in particular.

The remainder of this paper is organized as follows:

Section 3.2 present some background to the area, after which Section 3.3 describe the problems directed in this paper. In Section 3.4, we present our proposed solution to these problems. In Section 3.5, we describe an implementation of the proposed method that was a part of a recent case-study. Section 3.6 provides a short survey of related work. The paper is concluded in Section 3.7, where we also provide some discussions on future work.

3.2 Background

The general idea behind deterministic replay is to record (to *monitor* and *log*) sufficient information about a *reference execution*, typically one that ended in a failure, of the non-deterministic system to facilitate *replay*. Replay can be described as the production of a facsimile of the reference execution based on that recording. While cyclically replaying that *replay execution* of the system it is possible to cyclically debug the system.

The level of detail and length of the recording defines the accuracy of the replayed (facsimile-) execution relative to the reference execution. Which level of accuracy required is in turn dictated by the *fault hypothesis*, i.e. what type of bugs do we assume may exist, and the infrastructure, i.e. what type of bugs are possible [14]. The more intricate bugs we assume can exist in the system, the more information we need to record. By using our method of recording and deterministically replaying executions, we extend the sequential failure fault hypothesis to also include ordering-, synchronization and timing failures [14].

Recordings are facilitated by inserting probes into the system. These probes will produce auxiliary outputs, buffered into logs during the reference execution. Probes can be implemented in hardware, software, or be a hybrid of those two. The difference between the approaches is essentially defined by the amount of perturbation introduced, i.e. clock cycles consumed or amount of memory used. If probes are added, removed, or altered over time, so that the level of perturbation varies, the system will suffer a *probe effect* [3], which may change the behavior of the system - and prove counterproductive (as it will invalidate previous verification efforts). Thus, probes that incur significant perturbation should be left permanently even after deployment [18]. For more elaborate discussions on this see [4, 14]. The fact that the overhead incurred by the probes should be accounted for in the schedulability analysis also suggests that the overhead should be deterministic and limited.

A recording consists of two parts [10]:

- the *data-flow* describes variations in local and global variables, as well as inputs or output used by the task set, while
- the *control-flow* describes alterations in the execution, e.g., scheduled preemptions, blocking system calls, or interrupts.

Together with the system source code, this information defines the execution of the system.

For post-mortem debugging of embedded systems, the information held

by the recordings is essential for a successfully produced replay execution. As these systems have long up-times, if this information was to cover the entire execution it would consume large quantities of memory. However, a trait of these systems is that the amount of resources (temporally and spatially) available is limited. In order to minimize the amount of memory needed for a recording that captures a reference execution it is possible to apply infinite loop recorders to a system based on finite length cyclic arrays. The problem to solve, however, is how to start the system and make it behave like during the reference execution? This problem can also be described by the two questions formulated in Section 3.1. Previous related work has not provided us with answers to this.

3.3 Starting points for replay executions

To deterministically set up the production of a facsimile - to setup a deterministic replay of a recorded reference execution - we need to do two things:

- First, correlate the recordings and identify potential starting points for each task. A starting point consists of a control-flow event and a corresponding and sufficient beginning condition (state, message-body, etc) in the data-flow recording. (The data-flow entry is named *saturated point*, as it describes a sufficient data-state of the individual task.)
- Second, it is necessary to re-execute the restarted/rebooted target system to a point in the program, typically a potentially blocking system call that matches a saturated point in the recording. From this point onwards, the target system can subsequently be deterministically re-executed by a replay-mechanism until the end of the recording.

3.3.1 Finding starting points in the recording

To define a *starting point* for a replay execution it is required that we have a sufficient set of accurate information for replay of the reference execution at the time of that starting point. That is, the recording at that instant must have captured the sufficient conditions for a specific task instance, such that from that instant it is possible, with the remaining information in the recording, to re-execute the system to the end of the recording (e.g. the failure); switching tasks in and out, substituting the contents of state variables, messages, and peripheral inputs with the recorded values. Starting points are defined by the

cut set of the data-flow and the control-flow. Thus, there must, for any valid starting point in the control-flow recording, exist a corresponding beginning condition (state, message, etc) in the data-flow recording. Figure 3.1 illustrates this cut set.

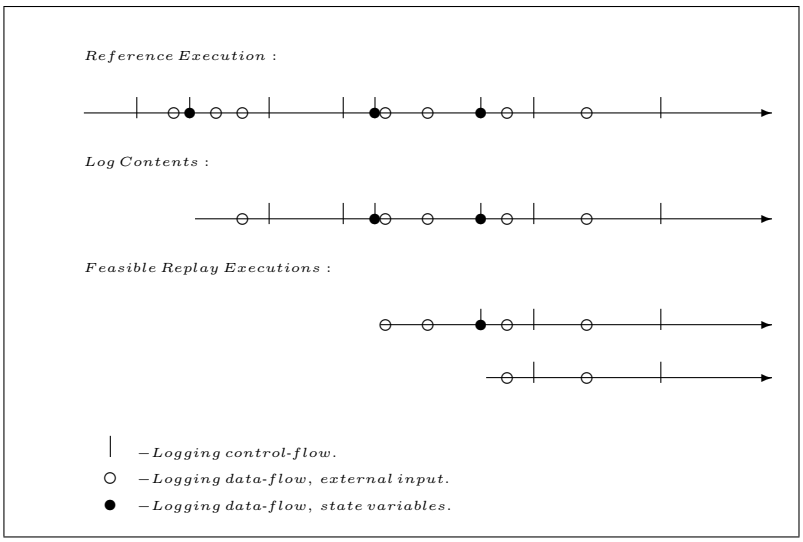

Figure 3.1: Replay execution based on log from a reference execution.

As the size of the data that describe the full context of any given task is usually substantial, it is not feasible to allow the replay execution to start at an arbitrary point in the reference execution (e.g. at preemptions or interrupt hits) since this would entail saving the entire task context. We select a set of points from which the start of replay can begin: viable starting points are task activation (first time) and blocking system calls. This does not mean that we cannot reproduce preemptions or interrupts, only that we cannot start at such events.

3.3.2 Finding starting points for the replay execution

Using some debugging infrastructure, e.g., an ordinary interactive debugger or a breakpoint interface in the real-time operating system, breakpoints should initially be set for all potential starting points: typically all blocking system

calls and task entries in the restarted/rebooted target system. This is usually sufficient since preemptions or interrupt hit points are not valid starting points. Each task in the target system is started with the same parameters as during the reference execution. When, eventually, a task makes a system call, it will hit a breakpoint. As the execution of the task is halted, all entries in the recording with that system call reference and task identity can be used as starting points for a replay. Thus, if at least one saturated starting point that matches the system call is found in the recording, the beginning conditions (e.g., message contents, variable contents, etc.) is substituted with the recorded values. When a starting point has been reached for every task (or desired sub set) in the recording, we can start replaying the system.

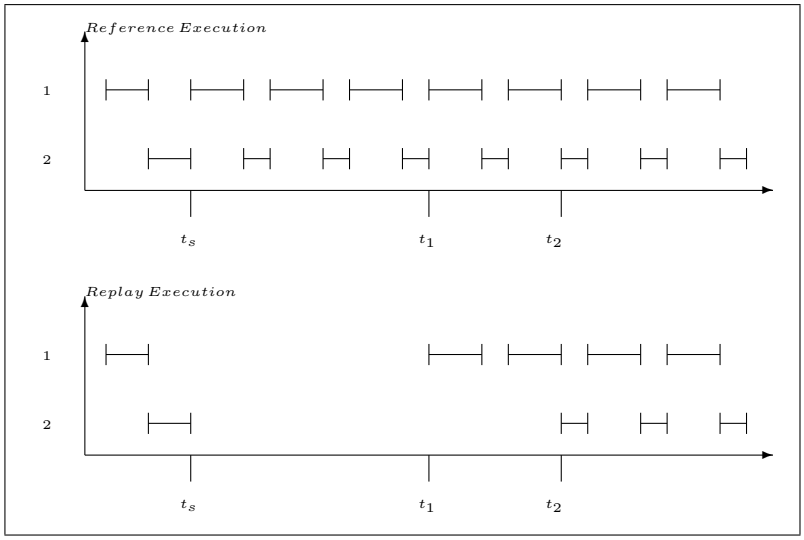

Figure 3.2: Execution-traces for reference and replay executions.

Note that, as we can see in Figure 3.2, the replay may begin at different times for different tasks. Replay of task 1 is initiated at time t_1, while replay of task 2 is not initiated until $t_2 > t_1$. In the span between t_1 and t_2, task 1 may complete a number of iterations. This implies that a replayed task, which requires input from another task in the set of replayed tasks, may be forced to rely on the contents of the accumulated log for the required input.

3.3.3 Replay

When the initialization is ready, the replay will step forward as the time index is incremented at each control-flow event that is successfully matched. In addition, if a subsequent preemption or interrupt event for the current task is found in the control-flow, its corresponding conditional breakpoint is set, making it possible to replay this event as that breakpoint is hit. Once breakpoints representing such asynchronous events as preemptions and interrupts are hit and successfully matched, they can be removed in order to enhance the performance of the replay session. Since we have eliminated the dependency of the external process in real-time and replaced the temporal and functional context of the application with the recorded data- and control-flow timelines, we can replay the system history repeatedly.

3.4 Starting point prerequisites

In this section, we define what constitutes a starting point for a deterministic replay execution in a real-time system.

3.4.1 Definitions and assumptions

We differ between a *global* and a *local* starting point. A local starting point is a starting point for a specific task, a global starting point is a set of local starting points which can be used as starting point for the set of tasks that are to be replayed. We define *system call references* to be calls to the same system call from the same program counter (PC) value, and *system call instances* to be incarnations of calls to the same system call but possibly from different references. The set of all system calls is labeled C.

For local starting points, we assume that:

We are able to incorporate probes into the operating system. Some of the probes must be simple *kernel probes* [14], i.e. integrated into the operating system. These receive some parameters from the operating system, and their execution is protected from interrupts.

System call references are monitored. The set of all monitored instances of any system call in C is labeled E. A call by a task from program counter value pc, to a system call $c \in C$ is denoted $e_n \in E$, where n is a unique and temporally ordered identifier for elements in E. Together with the entry, it is possible to store also a data-segment which is a subset of the tasks data-flow.

Interrupts, exceptions, and preemptions are monitored. The set of all

monitored interrupts, exceptions, and preemptions is labeled I. The union $F = E \cup I$ is the set of control-flow events.

A subset of the events in E are of a set E^s that can be used as local starting points, the entry points of these system calls are labeled *potential local starting points*, the set of which is denoted $C^s \subseteq C$. Simultaneously with the monitoring of potential starting points, the full data-flow is also monitored (e.g. state variables). Thus, a monitored event $e \in E$ is a 5-tuple $e = \langle n, c, i, pc, d \rangle$, where i denotes the task which was executing when the monitoring was performed.

The size of the set F is assumed to be large, some of the entries are later evicted from memory as the space available to store them in is relatively small. This eviction is performed by an *eviction scheduler* [5]. At the end of the monitoring session, $F_{log} \subseteq F$ denotes the set of entries that still remain in memory - which are still in the log.

A subset F^s_{log} of the events at potential local starting points is the set of local starting points. A local starting point is an event which is in E^s and which is still in the log.

A started task will always reach a potential local starting point without help from the replay engine or other external process outside the system.

The phase of initialization is deterministic for all tasks. When a system is restarted, there is a phase of initialization before the system reaches its first potential local starting point. That phase is deterministic.

We define a global starting point as a set of local starting points S where it is true that:

There is one and only one local starting point for each task per global starting point S. If there is more then one feasible local starting point, one is chosen.

The replay is not dependent on any irreproducible communication. Given an instance of a communication between two tasks, where the event e_n represents the act of transmitting a message and e_m the act of receiving the same. If the global starting point for the receiving task is prior to e_m, it is either true that the global starting point for the sending task is prior to e_n, or that one of the two events are still represented by entries in the log.

3.4.2 Finding starting points

Using an ordinary interactive debugger, we initially place breakpoints at all potential starting points. Each task to be replayed is started with the same parameters as during the reference execution. As a task calls a system call, such

that that system call is a potential starting point, it will hit its first breakpoint. As the execution of the task is halted, all entries in F_{log} with that system call reference and task identity can be used as starting points for a replay. Thus, if at least one such entry is found in F_{log}, the data-state d of the task is substituted for the data-state from one such entry, after which it is considered that the data-state of the task and the corresponding data-state of its predecessor from the reference execution are indistinguishable.

When a local starting point has been reached for every task in the entire set of tasks to be replayed, the global starting point S has been established.

Other schemes for replay have allowed such intermediate messages to be partially supplied by the replayed instance of the producing task [8, 20] by using *adaptive logging*. However, as previously published solutions make on-line decisions about whether to log or not to log a monitored event originating from high-perturbing software probes, there is an increase of the jitter in the system. Jitter will reduce the testability of the system [16], wherefore gains in time overhead for the logging-procedure must be balanced with respect to this. It would however be possible to make certain gains with regard to memory resources required, without compromising the testability, but that would require eviction strategies such as that presented in [5].

3.4.3 Multiple consecutive starting points

Above, we posted the assumptions that a started task will always reach a potential starting point and that the phase of initialization will always be deterministic. If we assume that tasks are constructed as control-loops; a setup sequence is followed by an infinite loop. This, together with a wish to always be able to replay a reference execution, leads to the requirement that the first feasible starting point must lie at the first instruction of the infinite loop. In addition, the setup sequence cannot be non-deterministic, wherefore it cannot operate on any semaphores or similar. These are clearly unfortunate limitations.

If we wish to have other task-constructs, such as the one in Figure 3.3, we must take additional steps to ensure the presence of local starting points in the log. This can be performed by ensuring that a subset of the collected pool of recordings is conserved in the recordings from the reference execution: If at least one entry of every feasible local starting point that is encountered during the reference execution is kept in the log, together with sufficient information to allow a replay to the next consecutive feasible starting point in the execution, we can allow more complex task-constructs.

Figure 3.3 shows a setup where both the states $S1$ and $S2$ can be used as local starting points. However, a recording that has spent too many iterations in $S2$ may no longer have entries from $S1$ in the log. Wherefore a starting point cannot be found. Thus, we must separate the logs that store entries from the two events. A simple approach could be to have separate circular queues for the two. This will ensure that entries that describe the transition from $S1$ to $S2$ are always accessible if they occurred during the reference execution. Thus, entries from $S2$ can always be replayed if they occurred during the reference execution.

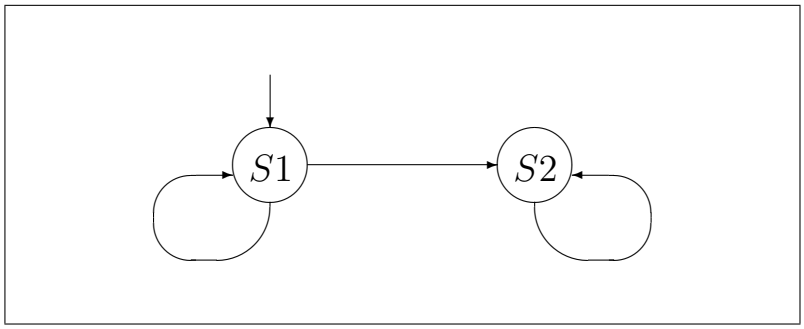

Figure 3.3: A task with a modechange.

Previously proposed schemes for logging data, have either been centralized circular queues [11], which is a FIFO-queue, or have such characteristics that they compromise testability (see Section 3.4.2 on adaptive logging). Hence, such methods cannot be efficiently used in this context.

3.4.4 Replay

When the global starting point S has been established, conditional breakpoints are set at all unique program counter values where events occurred such that they are in I, and also still in F_{log}. These breakpoints represents events that should be replayed, but have such properties that the address at which they occur is not deterministic.

Breakpoints are also set at the entry points of all system calls which are not potential starting points. Previously positioned breakpoints at the entry points of all potential starting points remain in place. Thereafter, the replay can be

commenced.

The replay uses the positioned breakpoints to control the preemption order implied by the entries in the accumulated log. Tasks at breakpoints are released in the pattern dictated by the control-flow. As system calls are encountered, for which there is a valid entry in the data-flow, that data is injected into the task at the correct points with respect to the control-flow.

As feasible starting points (which are references to potentially blocking system calls) are encountered, we can choose to start from another instance of that system call reference. This can be describes as jumping forwards or backwards in time.

3.5 Implementation

An implementation of this method was part of an industrial case study [12, 17], which aimed to achieve deterministic replay for post-mortem debugging of an industrial robot control system. The developer of the investigated system is among the largest industrial robot manufacturers in the world, ABB Robotics. Their system consists of several computing control systems, signal processing systems and I/O units. We applied our methods to a part of the system that consists of a plentitude of tasks, approximately 2.5 million lines of C code, and is run on the commercial VxWorks real-time operating system (RTOS).

As stated earlier, the complete data- and control-flow, together with the application code defines an unique execution of the application. In our implementation, control-flow and data-flow are monitored separately by the use of software probes inserted in the application code and in the kernel. Although more elaborate schemes have been proposed [5], we use basic cyclic buffers for system control-flow and data-flow logging.

3.5.1 Data-flow recording

The data-flow probes are made up of simple monitoring functions, called within the code of each task. During the reference execution, when called, these probes store the values of selected static variables, messages received, or external sensors read. During the replay execution, however, this operation is reversed, such that the information is read from the data-flow log rather than being stored onto it. As the replay execution is also executing the deterministic phase of initialization, we do not have to record the state of variables that are part of the parameterization. The selection of which data to store/retrieve at

each data-flow probing is managed by the use of data filters, defined by the developer.

3.5.2 Control-flow recording

As for the control-flow probes, these are less application-specific but much more kernel-bound. Since VxWorks does not ship with complete source-code (yet), we have made use of the kernel hooks included in the RTOS. Using these, code can be inserted for execution in task switches, interrupts, and other kernel events. These hook probes, combined with a set of system call wrappers, allow us to instrument all task switches in the sense of determining their cause, internal ordering and location of the occurrence.

3.5.3 Correlating data- and control-flow

To be able to perform a replay of the reference execution, the data-flow and the control-flow logs need to be correlated. For example, a local starting point $e^s \in F^s_{log}$ is made up of a log point where the control-flow and data-flow entries for that task coincide.

```
TaskA()
{
   int gvar = 0;

   while(FOREVER)
   {
      msgQReceive(msgQId, &msg,
                  maxNBytes, timeout1);
      probe(MSG_PROBE);
      .
      subr(gvar);
      gvar++;
      .
      semTake(sem, timeout2);
      .
   }
}
```

Figure 3.4: Probed code example.

Consider, for instance, the example code in Figure 3.4. The potentially blocking system call `msgQReceive` is followed by a software probe, storing (or retrieving) the contents of the received message. In addition, the value of the global variable `gvar` should be stored due to the fact that it helps define the state of the task in each iteration of the loop. If the global variable is not stored, the replay execution will always start with a `gvar`-value of zero, corrupting the correlation between data- and control-flow of the reference execution facsimile.

In the case of an empty message queue, the task will make a transition to a waiting state and thus cause a task switch, which will be logged as an entry in the control-flow log. When a message arrives to the queue, the task will be awakened and the software probe will execute, storing the received message in the data-flow log. This is an example of a situation where control- and data-flow log entries coincide, producing a potential local starting point for this task.

On the other hand, look at the next potentially blocking system call, `semTake`. When executed, if the semaphore is taken, this call will cause a running- to waiting- state transition for this task as well. This transition will be stored in the control-flow log and will be essential for the deterministic reproduction of the execution. However, since no data-flow is stored in conjunction with this, the task state cannot be restored during the replay execution at this location and the control-flow log entry is not part of the potential local starting point set, F^s.

3.5.4 Starting the replay execution

As stated earlier, the replay execution is initiated by breakpoints being set at all potential local starting points in the code of the system. In VxWorks, this is done by issuing breakpoint commands to the on-target debug task. Once these breakpoints are set, the system application can be started and executed up until all tasks have hit their first breakpoint. This will leave the entire application in a suspended state, from which we are able to chose, from local starting points in the log, which task to release for execution first. The chosen task is released and deterministically executed up until its next breakpointed location of task interleaving (blocking system call, preemption or interrupt) in the log. Reaching this location might call for enforcing of synchronization mechanisms, such as semaphores or message queues that did not block during the reference execution. At this point, a new selection is made, based on the log sequence, about which task to choose for execution.

3.5.5 Concerns about the reproduction of inter-task communication activities

By viewing each task in the replay execution as an fairly autonomous and isolated entity, we ensure that the state consistency of the global starting point does not depend on any irreproducible communication. In our implementation, a message sent by a task *A* to a subsequent task *B* is logged using a data-flow probe in the execution of task *B*. Using this approach, tasks *A* and *B* operate in isolated environments during the replay execution and task *B* does not have to rely on the correct deliverance of messages from task *A* in order to be reproduced deterministically. However, monitoring all inter-task communication explicitly might be a time-consuming and expensive activity and a more thorough analysis of the system task execution behavior could let us identify periodic transactions of tasks, within which some messages can be assumed to be reproducible during replay [9]. We have chosen not to exploit this fact, which may allow a reduction in the overhead from the monitoring activities, as the jitter of current technologies [9] will compromise the testability of the system [16].

3.6 Related work

With respect to related work in the field of replay debugging of concurrent programs and real-time systems most references are quite old. Recent advancement in the field has been meagre. On the special topic of finding starting points for replay of real-time systems, no comprehensive studies have been published hitherto. The only work known to have some similarities [8, 20] is limited to replay of message passing in concurrent software, and does not cover real-time issues like scheduled preemptions, access to critical sections, or interrupts. Also, the jitter of these solutions causes the testability to be compromised.

On the general topic of deterministic replay previous work published has either been relying on special hardware [2, 19], or on special compilers generating dedicated instrumented code [2, 7]. This has limited the applicability of their solutions on standard hardware and standard real-time operating system software. Other approaches do not rely on special compilers or hardware but lack in the respect that they can only replay concurrent program execution events like rendezvous, but not real-time specific events like scheduled preemptions, asynchronous interrupts or mutual exclusion

operations [1, 13, 20]. For a more elaborate discussion on related work see [4]. Earlier versions of our deterministic replay technique, which supported replay of interrupts, preemption of tasks and distributed transactions, have been presented previously [14, 15, 17]. However, none of those papers elaborated on how to identify starting points.

3.7 Conclusions

In this paper, we presented a method for initiating a replay execution based on a previous reference execution.

The replay execution can, deterministically, be cyclically repeated, it is possible to stop the execution by inserting breakpoints at arbitrary positions, and variables used can be inspected. It is therefore possible to use the replay execution when cyclically debugging non-deterministic real-time systems.

Previous work with replay has not been concerned with the problem of initiating the replay execution; to our knowledge, the method presented here is the only known to this date.

3.7.1 Future work

In Section 3.4.3, we described a simple solution to the problem of allowing replay of more complex task structures then simple control loops. In our future work, we will elaborate on this, and investigate solutions based on the logging structure that we presented in [5].

We will also direct the issue, described in Section 3.5.5, of re-executing rather then logging intermediate messages.

3.7.2 Acknowledgements

The work presented in this paper was supported by the Swedish Foundation for Strategic Research (SSF) via the research programme SAVE, the Swedish Institute of Computer Science (SICS), and Mälardalen University.

We would like to thank Ingemar Reiyer and Roger Mellander at ABB Robotics for the opportunity to validate the method presented here.

Bibliography

[1] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A pertrubation-free replay platform for cross-optimized multithreaded applications. In *In Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, April 2001.

[2] P. Dodd and C. Ravishankar. Monitoring and debugging distributed real-time programs. *Software - Practice and Experience*, 22(10):863–877, October 1992.

[3] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[4] J. Huselius. Debugging parallel systems: A state of the art report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[5] J. Huselius. Logging without compromising testability. Technical Report 87, Mälardalen University, Department of Computer Science and Engineering, December 2002.

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

[7] J. Mellor-Crummey and T. 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.

[8] R. Netzer and J. Xu. Adaptive message logging for incremental program replay. *Parallel & Distributed Technology*, 1(4):32–39, November 1993.

[9] R. Netzer and Y. Xu. Replaying distributed programs without message logging. In *the 6th International Symposium on High Performance Distributed Computing*, pages 137–147, August 1997.

[10] B. Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

[11] D. Stewart and M. 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.

[12] D. Sundmark, H. Thane, J. Huselius, A. Pettersson, R. Mellander, I. Reiyer, and M. Kallvi. Replay debugging of complex real-time systems: Experiences from two industrial case studies. In *Proceedings of the* 5^{th} *International Workshop on Automated Debugging*, pages 211–222, September 2003.

[13] K.-C. Tai, R. Carver, and E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):280–287, Januari 1991.

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

[15] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, pages 265–272. IEEE Computer Society, June 2000.

[16] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[17] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 288–295, April 2003. Presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD03).

[18] J. Tsai, Y. Bi, S. Yang, and R. Smith. *Distributed Real-Time Systems: Monitoring Visualization and Debugging and Analysis*, chapter 3.1, page 51. Wiley-Interscience, 1996.

[19] J. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[20] F. Zambonelli and R. 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.

Chapter 4

Paper C: Recording for Replay of Sporadic Real-Time Systems

Joel Huselius and Henrik Thane. A version of this paper has been submitted for publication.

Abstract

Deterministic replay is a technique that allows cyclic debugging of multi-tasking real-time systems, the technique requires costly recording of data during system execution. This paper presents an implementation and an evaluation of a logging structure intended for use when recording the execution of a sporadic real-time system using a limited memory space. The resulting log can be transparently used by a known replay-mechanism to deterministically reproduce the executions - a key requirement for cyclic debugging. We motivate and define the different requirements on logging structures used for recording executions of sporadic real-time systems, and show by simulation that the presented logging structure outperforms the only previously known competitor.

4.1 Introduction

Cyclic debugging is the commonly used term for the process of debugging a system using an ordinary debugger (e.g., using gdb). It is an iterative process that restarts the system over and over again (with the same input) to pinpoint the bug, hence "cyclic". A requirement for successful cyclic debugging is a repeatable execution - a requirement that some concurrent systems, real-time systems, etc., cannot fulfill.

Reproduction of execution behavior through *replay* has previously been presented as plausible means for successfully realizing cyclic debugging of systems that incorporate elements of non-deterministic behavior and/or time-dependence (e.g. real-time systems) [8, 9, 12, 16]. By *recording* observed and extracted information that describe an execution of a system (hereafter referred to as *to record an execution*, or *the recording effort*), that *reference execution* can be recreated deterministically offline by replaying the elements of non-determinism. These *replay executions* can then be debugged using a standard integrated development environment (IDE), a recent publication [11] even showed the applicability of the method in industrial state-of-practice real-time systems.

One of the main issues with replay is the memory requirements of implementations; recording an entire execution often requires large amounts of memory. If the memory resources available to the recording effort are considered to be small in relation to the size of the data that must be logged, these systems must allow the replay to start from a state other then the initial state of the system. Previous work has shown [11, 15] that this can be performed by taking memory excluding checkpoints [7] from where the system can be restarted. However, it is then not certain that a replay execution can be performed for any reference execution [4]. In this paper, we present a method that can guarantee the success of replay even when memory space is sparse. In addition, we will show by simulation that our method outperforms the only known competitor.

The organization of the remainder of this paper is as follows: Section 4.2 will provide a background and a motivation for the work presented in the following sections. Following, Section 4.3 will present a system model and the sparse related work found in literature. The contribution of the paper is presented in Section 4.4, after which the paper is concluded in Section 4.5.

4.2 Background and motivation

The general idea behind replay is to, by inserting *probes* into the system, *record* (to *monitor* and *log*) sufficient information about a *reference execution* of the non-deterministic system to facilitate the offline reproduction of a *replay execution*. The replay execution can thus be described as a facsimile of the reference execution. As the replay execution has a *virtual time*, and all inputs are recreated deterministically, the execution can be examined and halted without intruding on the system functionality - thus, the system is repeatable, and can be subjected to cyclical debugging.

Implementation strategies for probes are: software (SW), hardware (HW), or some hybrid (HY) of those; implementation strategies can be compared with respect to the perturbation-cost (SW - high, HW -low), economical-cost (SW - low, HW - high), and portability (SW - high, HW - low).

We assume that probes are implemented in software, and that they are run on the same nodes as the rest of the system. Also, we assume that *deterministic replay* [11, 15], a recent implementation of the replay strategy using a commercial-off-the-shelf (COTS) IDE, is used. The implementation does not assume that the log from the reference execution describes the reference execution in its entirety; by using memory excluding checkpoints, the replay is allowed to start from a state other then the initial state of the system, which implies that some sequences of the log may be discarded before completion of the reference execution. In this paper, we are concerned with the *logging structure* that takes the decisions of which information to keep; the logging structure is the algorithm that discards older log records for newer ones.

4.2.1 Starting replay

A recent publication by Huselius et al. [4] presents a method for initiating a deterministic replay together with the system requirements for that method. This method is currently the only one known to us that discuss this issue. From the system requirements, it was deduced that the logging structure plays an important role in guaranteeing a successful replay; without a suitable logging structure, if the code of the system deviates from a simple control-loop-model, it is not guaranteed that sufficient information of the reference execution remains to allow a correct replay.

Basically, the method restarts the system by treating each task independently, this will lead to a situation where tasks may start their replay at different points in the virtual time of the replay execution. However, the

replay cannot be considered to be correct until all tasks are started. We name the interval in which all tasks are concurrently replayed: the *shortest interval of replay* (SIR).

The method for starting a replay stipulates that, for each task in the system that is to be replayed, *potential starting points* are identified in the code offline as a part of the monitoring effort. During the reference execution, memory excluding checkpoints [7] of the taskstate are taken at these locations. In order to replay a taskset, the individual tasks are restarted from scratch with the same input as during the reference execution. As a task reaches a potential local starting point, its context is replaced according to the log (thus, the potential starting point is indeed a starting point). When all tasks have encountered such starting points, the replay can continue as dictated by the log [4]. The first potential local starting point encountered during an execution, for which there are useable entries in the log, is called the *used starting point*.

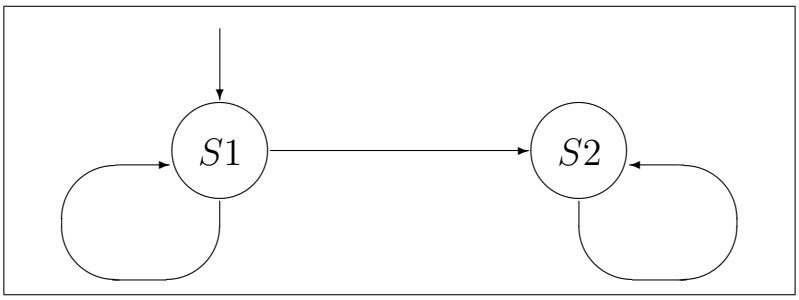

Figure 4.1: A task with several potential starting points.

One of the requirements posted by this method is that the execution of the individual task, up until it reaches the used starting point, is deterministic. Depicting the task as a finite state-machine (FSM), where potential local starting points are vertices and the possible execution paths between them are edges, we can see a problem illustrated by the example in Figure 4.1: In preparation for the replay, the task will be restarted after which it would end up in state $S1$ from where the replay must be started (remember: the execution must be deterministic until the first encountered starting point). However, the reference execution had to manage information from two monitored starting points - what if no information remain that describe a taskstate valid in $S1$? Say that all available information in the log describe taskstates valid in $S2$ - as

the information about $S2$ cannot be used until the replay execution has been shepherded there from $S1$, that would result in a situation where the replay cannot be performed.

Based on the above discussion, we formulate the following requirement: the logging structure must ensure that sufficient information is available in the log at the end of the reference execution to enable the replay to use or reproduce also the most recently monitored entry during the replay. We note that this is difficult if the FSM of tasks in the system are *non-deterministic* (using the definition of non-determinism by Milner [6]), but that the appropriate logging structure can ensure that the stated requirement will be fulfilled.

4.2.2 Length of replay

As Thane noted [12], theoretically, the SIR must cover the period of time from the infection of the system (the execution of a bug) to the failure of the system (when the presence of the executed bug is sensed by the system-environment). We call this period the *incubation period* of the system. In practice however, as we cannot know exactly which bugs are present in the system, the worst case incubation period of a system is not quantifiable. It is therefore the amount of memory assigned to the recording effort, and the rate with which this memory is effectively used, that limit the length of the SIR for a given system.

Thus, the goal of the logging effort must clearly be to extend the SIR for a particular execution of a given system as far as possible with the resources available. Hence, we can use the length of the SIR as a measure of the logging efficiency.

4.2.3 Contributions

The logging structure presented here has the ability to guarantee the possibility to perform replay of implementations that have a network of potential starting points as described in Section 4.2.1. Further, we show by simulation that the presented logging structure outperforms the only known competitor when monitoring sporadic real-time systems.

4.3 Logging

As argued by Thane and Hansson [13]: An executed instance of a multi-tasking system can be seen as a sequential program, an *execution scenario*

is a serialization of a multi-tasking system into a single-tasking system. Thus, a multi-tasking system can be seen as a set of single-tasking systems, each of which must be tested separately. The number of possible serializations of a system is increasing with the amount of jitter, as each execution scenario must be tested separately, jitter will increase the required testing effort. Hence, jitter compromises testability.

Based on this discussion, we conclude that an important constraint on a logging structure concerns the execution time of its implementation: In order not to compromise testability, the implementation should not introduce any additional jitter into the system [2]. One way to ensure this is to show that the implementation of the logging structure has a constant execution time.

Due to the effects of cashes, pipelines, etc., a constant execution time should be supported by a suitable hardware platform such as that presented by Delvai et al. [1]. However, the negative effect of jitter on testability has exponential characteristics [14], which leads to that even small reductions in jitter will have significant impact on testability.

4.3.1 Related work

Surveying the field of available logging structures, one finds that only a few alternatives are available today: As noted by Stewart Gentleman [9], the commonly used solution is a First-In-First-Out (FIFO) logging structure. There is also a method called *adaptive logging*, presented by Zambonelli and Netzer [16], but the online decision-maker of that method has substantial jitter-properties that will compromise the testability of the system it is used in. Sultan et al. presented a lazy garbage collection for distributed systems [10]. As the checkpointing process does not require a distributed solution to facilitate replay [4], and the checkpointing and garbage collection of their proposed method is transparent and unrelated to the application (thus inferring jitter in to the system), their method is not adaptable to the situation described here. A method presented by Huselius [2], on which this work is based, is called the Constant Execution Time Eviction Scheduler (CETES). CETES is an immature method that, since it requires all entries to be of the same size, cannot be efficiently used in real-world systems.

The relative recent interest in logging schemes [4], and the fact that not all replay-methods assume a limited memory, could be the reason for the small number of proposed alternatives. In embedded real-time systems however, the limited memory is indeed a reality.

4.3.2 System model

We assume an instrumented, preemptive, sporadic, online scheduled, multi-tasking real-time system where probes are implemented in software and are allowed to have their execution protected from interrupts. Tasks can communicate with each other and with the environment. Each task emits a single new *job* at a time, with a periodicity varying in the interval described by the task deadline and *sporadicity*. A new job of a task cannot be emitted prior to the completion of a previous.

The recording effort probes control-flow events (such as context-switches) and data-flow events (such as communication and checkpoints of task data-state). Each *event* observed by a probe results in an *entry* that is logged in memory in one or more *records*. The recording effort is assigned a memory-space on which both the implementation of the logging structure and the logs are to be kept (thus, smaller implementations will have larger space to keep logs).

4.3.3 FIFO logging structures

Concerning the FIFO scheme, three approaches seem feasible; either, Global FIFO (FIFO within the system), or Local FIFO (FIFO within tasks), or Starting point FIFO (FIFO within starting points). In Global FIFO (GFIFO), all memory available for logging is partitioned into a single FIFO-queue. In Local FIFO (LFIFO), each task will log entries to a dedicated queue, a separate queue is reserved for entries relevant for all tasks. In Starting point FIFO (SFIFO), each starting point will log entries to a dedicated queue, a separate queue is reserved for entries relevant for all starting points. An implementation of the LFIFO scheme is found in a previous publication [3].

Obvious difficulties with the three alternatives are as follows: a GFIFO-alternative must be able to accommodate several different sizes of entries, an LFIFO- or SFIFO- alternative must (based on entry sizes and arrival frequencies) calculate relative memory-requirements between queues.

Further, as GFIFO cannot distinguish between entries at all, the correct starting of a replay is not guaranteed (see Section 4.2.1). As GFIFO does not allocate memory to specific tasks, it cannot be guaranteed that all events from a certain probe will be represented in the queue; probes may be starved with respect to logging capacity. Thus, GFIFO is clearly unsuitable in this context and will not be investigated further in this article.

In the same way that GFIFO cannot serve a plentitude of tasks, LFIFO

cannot serve a plentitude of starting points. In order to use a FIFO solution as a logging structure for tasks with more then one starting point, the FIFO must be within starting points (SFIFO). However, assuming a task with several non-deterministic transitions between potential starting points, similar to that described in Figure 4.1, it is not possible to efficiently partition the available memory over the queues. Thus, SFIFO is clearly unsuitable in the context of this paper, and will not be investigated further in this article.

It seems that, assuming tasks with networks of potential starting points as described in Section 4.2.1, a dynamic logging structure is required.

4.3.4 CETES logging structures

A previous publication [2] introduces the Constant Execution Time Eviction Scheduler (CETES), a novel logging structure that allows dynamically adjustable distribution of memory resources to individual logging efforts. The algorithm maintains the LFIFO- and SFIFO- concept of queues, but allows the lengths of these to vary based on the contents of the log.

In addition, CETES (and ECETES) provide the developer with a tool that provide online control of the log-content without modifying the execution time of the application; a set of constraints are available to control the lengths, temporal span, and priority of each queue [2, 5]. These can, for example, be used to ensure that a particular queue will always contain all logged entries with in a temporal interval. They can also be used to switch on and of the logging of entries inserted to a queue (the perturbation will always stay the same though).

For a given entry size, the sequence of instructions executed is deterministic (see Table 4.5), which will result (assuming the SPEAR hardware) in a constant execution time for a given entry size. As a result, the execution time of the implementation is constant for monitoring a given entry, wherefore the jitter of the system is not increased, and the level of testability is therefore maintained.

However, the CETES implementation requires that all entries are of the same size, and the possibilities to control the properties of the queues are limited. The Extended Constant Execution Time Eviction Scheduler (ECETES), briefly described in [5], is introduced to remedy these issues. From the CETES-perspective, the primary improvement that comes with ECETES is the possibility to log also entries with varying sizes. In this article, we present an evaluation of this improved logging structure.

4.4 ECETES

The CETES and ECETES algorithms have a common basic functionality: Logging an entry of a number of records is preceded by the act of evicting the same number of records from a single queue. The output from the eviction process is a consistent chain of emptied record-bins that are then filled with the new entry. The process of eviction is the core to the functionality, and is identical in both the implementations (the eviction process is described in previous publications [2, 3, 5]). The basic structure of the eviction process is as follows: each available queue is examined only one time. Based on the examination process, the queue that is judged to loose least in terms of replay time from being subjected to eviction is chosen.

The source code to the current ECETES implementation is found in a recent technical report [3]. Compared to CETES, enhancements are as follows:

Entry sizes are allowed to vary between entries, even in the same queue. This is performed by implementing a 1-to-N mapping between entries and records; each entry can consist of one or many fixed-size records. The CETES-structure has, in order not to compromise testability, a constant execution time; each entry takes the same time to log, independent of queue-state or entry-state. As ECETES allows entries originating from different probes to have different sizes, logging two arbitrary entries may not take a constant time, but logging two entries of the same size always do. Figure 4.2 describes the assembler instruction-flow of the implementation, each loop in the function is performed a fixed number of times for an entry of a given size. Assuming that an entry produced by monitoring a given event always will produce an entry of the same size, this is a sufficient condition for maintaining the system testability.

The above mentioned examination each queue is performed by comparing queue properties and the properties of one of the records in each queue; CETES compared the next-to-last records of each queue, but ECETES does it a bit differently: If n records are to be evicted, the $(n+1)$:th record counted from the end of the queue is examined. This way, the process of eviction will identify a consistent chain of records in a single queue, and evict the records in that chain.

This is not an optimal solution, but it is the only working solution known to present day. The constraints on the execution time of the implementation limits the possibilities to realize a solution that is optimal according to the criteria defined in Section 4.2.2. We are thus forced to settle for a heuristic. A more efficient method would be to examine all records in the pool, and decide not on a whole consistent chain of records to evict, but hand-pick every single record.

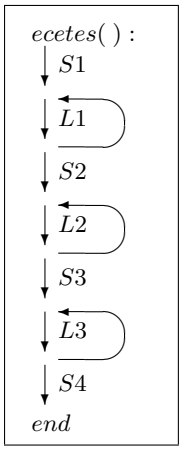

Figure 4.2: Instruction-flow of the ECETES-implementation (see Table 4.5 for labels).

However, in any real system, such an approach would inflict too heavily on the execution time of the algorithm.

As entries of different sizes (originating from different probes) can be logged in the same queue, all data-flow for one task can be logged in the same queue. When starting a replay, the first entry that will be of importance is an entry describing a checkpoint of the task-state. Thus, all records that are not part of a complete checkpoint, or has an older complete checkpoint in the queue can be marked as unneeded. (In the current implementation, ECETES will mark only unneeded records that are not part of any checkpoint.) As unneeded records can be evicted without penalty, these are considered first by ECETES.

4.4.1 Example

See Figure 4.3, where we have as system of two ECETES-queues (A and B). Each queue receives entries to log into records, and has a vector of logged records associated to it. There is a header for each of the queues ($H\ A$ and $H\ B$), these contain the constraints posted on the queue (see Section 4.3.4), and pointers to the first and last element in its vector of records. Also each record contains some information concerning its age and information about where to locate its neighbors.

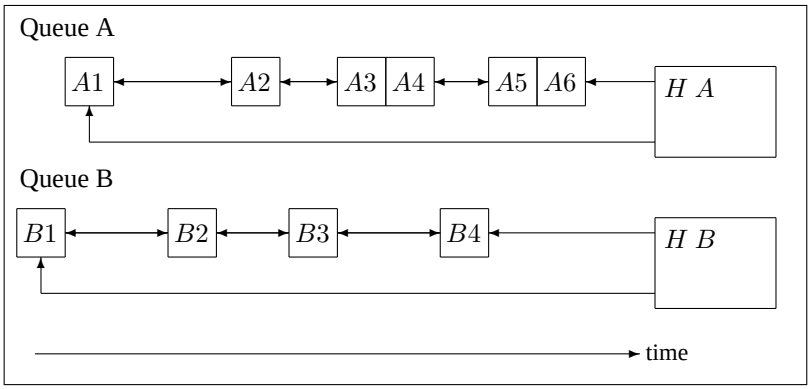

Figure 4.3: ECETES example.

Now, imagine that we have a new entry consisting of two records to store in queue A. Before storing the new records, we must identify a queue from where to evict two records. This identification is performed by examining the queue-headers, and one record from each queue ($A3$ and $B3$). These records are picked as they are the oldest records that will remain in the respective queue if two reocrds are evicted. Essentially, provided that it will not lead to a violation of the queue-constraints, the queue to which the oldest of the two records belong will the subjected to the eviction. Here, $B3$ is the oldest, so B will have to surrender its two oldest records ($B1$ and $B2$). The two new records ($A7$ and $A8$) can then be inserted, and the new records are then linked with their designated queue.

4.4.2 Evaluation of the ECETES logging structure

Due to the dynamic properties of the queues in the structure, it is possible to guarantee replay of systems with multiple starting points while keeping an efficient memory utilization. Thus, the part of the motivation (see Section 4.2.3) described in Section 4.2.1 is fulfilled by the ECETES log structure. In this section, we shall present an evaluation that was made to ensure that also the other part of the motivation (see Section 4.2.2) was improved relative to the only other known useable log structure, LFIFO.

We evaluated the ECETES logging structure by comparing it to the LFIFO logging structure in a simulator, the simulation intended to validate the

hypothesis that ECETES outperforms LFIFO in sporadic real-time systems. The validation is performed by measuring, given a certain memory budget, which of the tested methods that will provide the longest shortest interval of replay (SIR) (see Section 4.2.1) in such a system.

The simulator

The simulator used was tailor-made for the evaluation, code to the implementation is provided in a previous publication [3]. The task model is a fixed-priority scheduled sporadic task, priorities are set according to rate-monotonic, and complies to the system model specified in Section 4.3.2. Although the simulator is capable of modeling inter-task communication (IPC), this feature was not used during the validation process.

Simulation

We performed simulations with three different tasksets, the properties of the sets used are as displayed in tables Table 4.1, Table 4.2, and Table 4.3. Each taskset was subjected to 23 instances of simulation with different values on the sporadicity of the tasks of the taskset. Each instance was simulated 100'000 times.

Properties	**Task** A	**Task** B
Deadline [time units (tu)]	2000	8000
Execution time [tu]	990 - 999	3960 - 3999
Data-state [bytes]	1100	1100
IPC's in [number]	0	0
IPC's out [number]	0	0

Table 4.1: Simulator taskset 4.1.

In the setup for both ECETES and LFIFO, for each task, a queue was assigned to log the data-flow, and the monitored checkpoints. One queue was also assigned to monitor the control-flow of the entire system. In the case of the LFIFO logging structure, if the tasks of the system have IPC-queues for communication, these would most likely have to be assigned separate queues as the LFIFO-implementation used does not support different entry-sizes in the same queue. The ECETES implementation can, as entries from different probes can have different sizes, allow such entries to be logged together with

the data-state checkpoint of the receiving task.

Properties	**Task** A	**Task** B
Deadline	4000	8000
Execution time	990 - 999	3960 - 3999
Data-state	1100	1100
IPC's in	0	0
IPC's out	0	0

Table 4.2: Simulator taskset 4.2.

The LFIFO logging structure requires some parameterization regarding individual queue-lengths, these are calculated as follows: Checkpoints of the data-state are taken at the start of each job. Each system call that may result in a preemption (e.g., `semaphoreTake()`) may, in the worst case, result in two context switches per execution. By counting the maximum number of potentially blocking system calls that may, in the worst case, be executed for each job, the worst-case number of preemptions under some period of the system execution can be calculated.

Properties	**Task** A	**Task** B	**Task** C
Deadline	2000	4000	8000
Execution time	990 - 999	990 - 999	990 - 999
Data-state	1100	1100	1100
IPC's in	0	0	0
IPC's out	0	0	0

Table 4.3: Simulator taskset 4.3.

However, the worst-case calculation of control-flow entries is very pessimistic. Further, it is also expected that the size of a control-flow message will be much smaller than a data-flow message [11]. But in this setup, as we use the same entry size for all entries, a gained control-flow entry for the ECETES algorithm would reflect unproportionally in the results; if all entries are of the same size, a gained control-flow entry for the ECETES can be directly transferred to a data-flow entry, but in a real system, it would probably take several control-flow entries to do the same. Thus, in our simulation setup, we chose not to consider the memory allocated for logging the control-flow of the

system as it would unjustly favor the ECETES logging structure.

Overhead

We can measure overhead in two ways; spatial overhead describes the amount of memory consumed by an implementation, temporal overhead describes the execution resources required to execute it. In the system model above we assumed a hardware similar to the SPEAR-example. However, the work on that platform has not yet resulted in a C-compiler. We therefore choose to display the spatial overhead measured on the Intel platform, the same platform used for the simulations.

Post	**TS 4.1**	**TS 4.2**	**TS 4.3**
ECETES queues	3#	3#	4#
ECETES records	72#	63#	102#
ECETES max records	3#	3#	3#
Assembler code	389	389	389
Queue headers	96	96	128
Entry index	36	36	48
Records	74752	66560	105472
TOTAL ECETES	**75273**	**66057**	**106037**
TOTAL LFIFO	**76214**	**66074**	**106662**

Table 4.4: Spatial overhead in bytes of the Intel implementation.

For the test cases described in tables Table 4.1, Table 4.2, and Table 4.3, the overhead of the current ECETES implementation on an Intel platform is as follows: The temporal overhead is described by Table 4.5, where *Iterations* describe the number of times a block of code is executed for each call to the function, and *Inst* (or *Instructions*) describe the size of each block in assembler code instructions as described by Figure 4.2. Note that the instruction count of loop $L2$ does not include the body of the function `memcopy()`, which is called from within that loop (also note that all calls to the function memcopy has the same size-parameter: the maximum record size). Regarding the iterations, *queues* is a constant describing the number of queues that are defined in the monitoring activity, *records* is the number of record required to log the entry, and *max records* is a constant describing the maximum number of records required to log an entry in the system.

Path		Inst	Iterations
Sequence	S1	26	1
Loop	L1	92	queues
Sequence	S2	43	1
Loop	L2	36	records
Sequence	S3	35	1
Loop	L3	34	max records
Sequence	S4	7	1

Table 4.5: Temporal overhead in assembler instructions of the Intel implementation (see Table 4.4 for constants).

The spatial overhead is described by Table 4.4, where *TS* is *taskset*. In the same table, also the corresponding LFIFO overhead is shown (for more information on the LFIFO overhead, we refer to the technical report that introduced the CETES logging structure [2]). Note that the test was configured in favor of the LFIFO logging structure; the spatial LFIFO overhead is strictly larger then the corresponding ECETES overhead.

Simulation results

A given taskset has a sporadicity variable which is used for all tasks in the set. For each of the above defined tasksets, we performed a suite of 23 simulation setups with varying sporadicity. For each setup, a total of 100'000 simulations where performed, accumulating a total of $3 \cdot 23 \cdot 100'000 = 6'900'000$ simulations. The length of each simulation was allowed to vary randomly between high numbers (relative to the size of the memory allocated for the recording) so that the logging structure had to prioritize between the entries logged in every simulation. Also the execution times of each issued job was randomized within an interval as described by the taskset specifications above.

For each simulation performed, the shortest interval of replay (SIR) was measured based on the log contents from each of the algorithms. The result from a simulation is the relation between the ECETES and LFIFO SIR's; a result of 100% describes a tie between the subjects of the evaluation, a result below 100% indicates a win in favor of LFIFO, a result exceeding 100% is a win in favor of ECETES.

The results of the evaluation are displayed in tables: Table 4.4 for taskset 4.1, Table 4.5 for taskset 4.2, and Table 4.6 for taskset 4.3. Each simulation

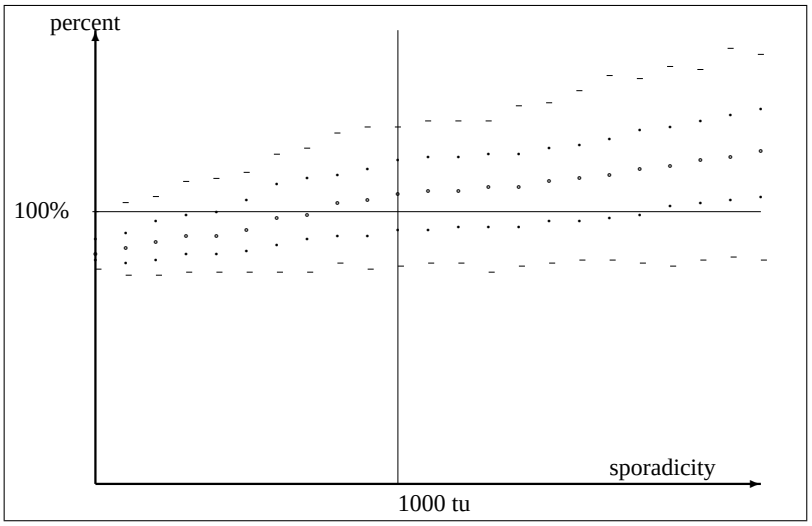

Figure 4.4: Simulation results using taskset 4.1.

suite has a column of five marks representing the five quartiles (Q0, Q1, Q2, Q3, and Q4, from the bottom and up) of the result. The graphs should be interpreted so that when Q2 has crossed the 100% barrier, the ECETES algorithm wins over, or out-performs, LFIFO in a majority of cases (assuming the current setting).

The profile of $Q0$-values maintains a close-to-constant value in a given taskset-simulation, thus indicating that the number of simulations performed for each simulation setup is sufficient to provide a sound result.

The simulation of taskset 4.1, accounted for in Figure 4.4, shows that ECETES outperforms LFIFO already at low levels of sporadicity relative to the periodicities of the tasks in the set (Q2 breaks the 100%-barrier at a sporadicity somewhere between 700 and 800 time units).

As we can see, this result is then further confirmed by the simulation of taskset 4.3, the simulation is accounted for in Figure 4.6: When a third task is added to the taskset, the sporadicity of the system execution is increased, and the ECETES performs even better in relation to LFIFO.

In order to concretize, we examine a particular simulation instance: With taskset 4.3, when the sprodicity is set to 1000 time units (tu), Task A has a

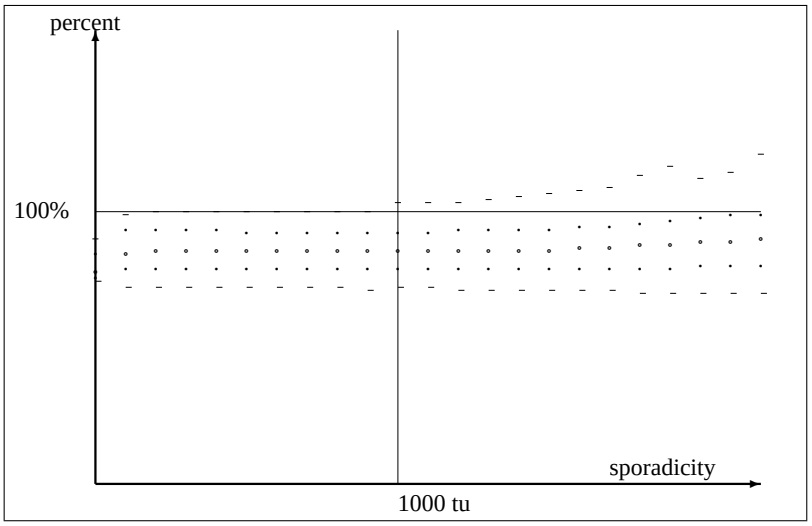

Figure 4.5: Simulation results using taskset 4.2.

periodicity in the closed interval [2000,2999] tu, the same closed interval for Task B is [4000,4999] tu, and [8000,8999] tu for Task C. In this particular simulation run, we measured that the Q2 SIR of ECETES was 39362 tu, and the same value for LFIFO was 37867 tu. Thus, the gain from using ECETES, in this particular simulation instance, seems clear.

In taskset 4.2, the simulation is accounted for in Figure 4.5, the periodicities of the tasks are spread in a larger interval of time, but the sporadicity constant varies in the same interval as in the simulations of the other tasksets. We can see that although ECETES eventually starts to gain on LFIFO, it does not out-perform it under the sporadicity interval simulated. As the tasks are issued with a lower frequency, ECETES requires larger amounts of sporadicity in order to out-perform LFIFO.

Concluding the report of the simulation results, our evaluation suggest that, as far as ECETES is concerned, a relation exists between the periodicities and the sporadicity of the taskset. The nature of this relation is however not known at present time.

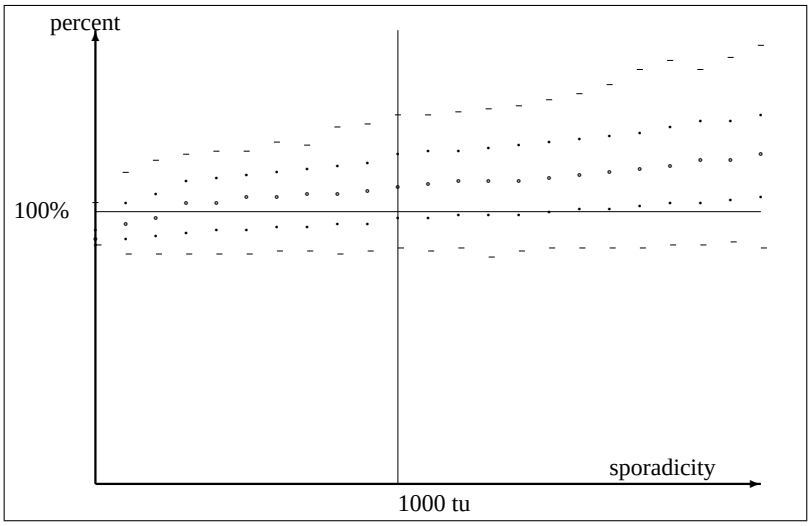

Figure 4.6: Simulation results using taskset 4.3.

4.5 Conclusions

It has previously been established that controlling the memory usage during recording of a reference execution is essential for the success of a following replay [4]. In this paper, we have presented the evaluation of an algorithm that can make efficient use of memory resources while guaranteeing the possibility to perform a correct replay based on a reference execution. Simulations presented here show that the proposed ECETES logging structure outperforms conventional FIFO-techniques in sporadic real-time systems.

4.5.1 Future work

There are still improvements that can be made to the proposed ECETES logging structure. These include improving the marking of *unneeded* records, and working on the selection technique which currently assumes the contents of the log prior to the insertion of the new record, a better performance would be reached would the resulting contents be considered instead.

Also the evaluation can be further improved, the effectiveness of the

ECETES is dependent of the relative sizes of logged entries; if these sizes are very similar, or at least multiples of each other, ECETES performs much better. This issue is not reflected in the validation performed here.

Last, the relation between sporadicity and periodicity must also be further investigated.

Bibliography

[1] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the SPEAR example. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 169–176, July 2003.

[2] J. Huselius. Logging without compromising testability. Technical Report 87, Mälardalen University, Department of Computer Science and Engineering, December 2002.

[3] J. Huselius. Source-code to the ECETES logging strategy. Technical Report, Mälardalen University, Department of Computer Science and Engineering, August 2003.

[4] J. Huselius, D. Sundmark, and H. Thane. Starting conditions for post-mortem debugging using deterministic replay of real-time systems. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 177–184, July 2003.

[5] J. Huselius, H. Thane, and D. Sundmark. Availability guarantee for deterministic replay starting points in real-time systems. In *Proceedings of the* 5^{th} *International Workshop on Automated Debugging*, pages 261–264, September 2003.

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

[7] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, 1999.

[8] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay and debugging. In *Proceedings of the* 4^{th} *International Workshop on Automated Debugging*, pages 5–18, August 2000.

[9] D. Stewart and M. 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.

[10] F. Sultan, T. Nguyen, and L. Iftode. Lazy garbage collection of recovery state for fault-tolerant distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 13(10):1085–1098, October 2002.

[11] D. Sundmark, H. Thane, J. Huselius, A. Pettersson, R. Mellander, I. Reiyer, and M. Kallvi. Replay debugging of complex real-time systems: Experiences from two industrial case studies. In *Proceedings of the* 5^{th} *International Workshop on Automated Debugging*, pages 211–222, September 2003.

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

[13] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[14] H. Thane, A. Pettersson, and H. Hansson. Integration testing of fixed priority scheduled real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, December 2001.

[15] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 288–295, April 2003. Presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD03).

[16] F. Zambonelli and R. 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.