IT Licentiate theses
2000-010
MRTC Report 00/25

Measurement and Simulation Based Techniques for Real-Time Systems Analysis

MARKUS LINDGREN

UPPSALA UNIVERSITY
Department of Information Technology

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

UPPSALA UNIVERSITY

Measurement and Simulation Based Techniques for Real-Time Systems Analysis

BY

MARKUS LINDGREN

December 2000

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2000

Measurement and Simulation Based Techniques for Real-Time Systems Analysis

Markus Lindgren
markus.lindgren@mdh.se

Mälardalen Real-Time Research Centre
Dept. of Computer Engineering
Mälardalen University
Box 883
SE-721 23 Västerås
Sweden

http://www.mrtc.mdh.se

ISSN 1404-5117
Printed by University Printers, Uppsala University, Sweden

Abstract

Rigorous methods for design and implementation of safety critical real-time systems are vital to avoid loss of human lives and/or severe economic losses. Unfortunately, many of these systems are designed and evaluated using ad-hoc techniques. There are, on the other hand, relatively well developed theories for modeling and analysis of timing and reliability. These theories are, however, seldom applied in industry for system development, mainly because of the simplifying model assumptions and lack of appropriate tool support.

This thesis presents two new methods aimed to narrow the gap between research results and industrial practice in evaluation and design of real-time systems.

The first contribution is a technique that can be used to derive worst-case execution time estimates for real-time software by measurments on the target system. Such estimates are essential when verifying if a system fulfills its timing requirements. The second contribution is a simulation based technique that can be used to evaluate timing aspects of distributed real-time systems, as well as calculating reliability estimates of these systems. Such estimates are essential in determining if a system meets its requirements sufficiently well. Compared to existing analytical methods for execution time analysis and schedulability analysis, which analyze models of the hardware, the starting point for both these methods are real target systems, rather than an abstract model with limited correspondance to reality.

The presented initial case-studies give clear evidence that the proposed methods have potential of being both applicable and useful.

Acknowledgements

There are several persons whom I wish to thank; without them this little creation would never have been completed. First of all I wish to thank my supervisor Hans Hansson, who has been supporting all my crazy (and sometimes a bit stupid) ideas. One also has to admire his efficiency and tremendous work capacity. Besides Hans, I would also like to especially thank Christer Norström, Sasikumar Punnekatt and Henrik Thane. Christer for his great insight and "helicopter"-view of the world, which has helped to bring things to their right places. Sasi for his help on the reliability paper and for his great skills with words. Henrik for stimulating discussions on various topics (not only research).

My other colleagues at the Department of Computer Engineering have also made this little journey quite pleasant, the greatest contributors to this are the people at the System Design Laboratory.

Tack *ska ni ha!*

Contents

List of Publications

The following articles are included in this licentiate[1] thesis:

A "Deriving Worst-Case Execution Time by Measurements"
By Markus Lindgren.
Technical Report 00/26, Mälardalen Real-Time Research Centre, Mälardalen University, November 2000. [Lin00]

B "Deriving Reliability Estimates of Distributed Real-Time Systems by Simulation"
By Markus Lindgren, Hans Hansson, Christer Norström and Sasikumar Punnekkat.
In Proc. of the 7^{th} International Conference on Real-Time Computing Systems and Applications; RTCSA'00, December 2000. [LHNP00]

C "Case-study: Using Simulation to Verify Real-Time Properties"
By Markus Lindgren and Christer Norstöm.
Technical Report 00/27, Mälardalen Real-Time Research Centre, Mälardalen University, November 2000. [LN00]

Besides the above articles I have written and published the following scientific papers/reports:

1. "Using Measurements to Derive the Worst-Case Execution Time"
   By Markus Lindgren, Hans Hansson and Henrik Thane.
   In Proc. of the 7^{th} International Conference on Real-Time Computing Systems and Applications; RTCSA'00, December 2000. [LHT00]

2. "System Development with Real-Time Components"
   By Damir Isović, Markus Lindgren and Ivica Crnkovic.
   In Proc. of ECOOP'2000 *Workshop 22 Pervasive Components*, June 2000. [ILC00]

[1] A licentiate degree is a Swedish degree halfway between MSc and PhD.

3. “Real-Time Components”
   By Damir Isović and Markus Lindgren.
   Technical Report. Mälardalen Real-Time Research Centre, Mälardalen University, March 2000. [IL00]

4. “Design and Scheduling of Shared Displays for Embedded Systems”
   By Hans Hansson and Markus Lindgren.
   In Proc. Swedish National Real-Time Conference SNART’99, August 1999. [HL99]

Chapter 1

Introduction

As processors year-by-year become more powerful, more reliable and less expensive, they also become an attractive alternative to be used in areas where not used before. For example, nowadays it is common practice to use micro-controllers to control many aspects of automotive systems; something that was not feasible not too many years back. In many cases, computers replace sub-systems that previously were controlled entirely by mechanical systems. But not only do computers replace existing mechanical solutions, they also have a potential of providing more functionality with higher reliability at less cost. This is, however, not always easy to achieve with current methods and tools.

The above type of applications have requirements which cannot be found in common desktop computing systems (PCs). One important class of such requirements are real-time requirements. These are often of the form: perform task x before time y, and/or do not perform task x before time z.

To further complicate matters, costs must often be kept at a minimum, implying that different application functions must *share resources*, e.g., computing resources. This sharing can also take place over networks of processors, so called *distributed systems*. In the research community *schedulabilty analysis* is used to determine whether a set of tasks (implementing a set of application functions) can be guaranteed to fulfill their temporal requirements on a set of resources, e.g., processors and networks.

Real-time systems are often classified into *hard* and *soft* systems. In hard real-time systems timing requirements *must* be guaranteed, otherwise catastrophic consequences may occur, for example, loss of human life or severe economic losses. Soft real-time systems *should* try to meet their timing requirements, otherwise performance will decrease. Hence, there is a difference in the criticality of meeting timing constraints in soft and hard systems.

In industry systems cannot as easily be classified into different classes, like soft and hard (black and white). Industrial systems may have real-time requirements which could be characterized as hard, but since there is always a tradeoff with other aspects, such as cost, the hard requirements are typically not strictly enforced. Hence, it is more appropriate to term these systems gray. Also note that, the end result of an industrial development is a physical system for which it in a strict sense is impossible to satisfy a hard requirement, simply because such requirements can only be proven to hold with some simplifying model assumptions[1].

This thesis presents two new methods for coping with real-time requirements. The first method deals with worst-case execution time of programs, and the second with modeling and analysis of system level timing and reliability. In the first method, measurements are used to find low-level timing of real-time programs/tasks on a hardware with an unknown timing model. Work on this is presented in paper A. The second method is a simulation based approach for determining the timing reliability of industry type systems, i.e., of gray systems. This work is described in papers B and C.

These two issues are not directly related, but both are essential for verifying timing properties of real-time systems. The first part derives low-level timing for a single task. The result is then fed to a high-level timing tool that computes the timing behavior of the task, which is an essential parameter in determining if the entire system can meet its timing requirements or not. The second part presents an approach for looking at system level timing, including how interactions among tasks affect timing. The two approaches can both be used for evaluation of soft as well as hard real-time systems. Compared to existing scientific methods we rely more on the systems themselves for making our predictions, hoping that this is more attractive to engineers and more applicable to industrial systems. (Schedulability analysis has not yet gained wide-spread use in industry.)

The proposed methods are useful at different stages of real-time system development. Using measurements to derive the worst-case execution times are to be used at later stages of development, when the system has been implemented and timing assumptions need to be validated. Simulation, on the other hand, is useful during the entire development process. At early stages of development abstract models of the system under design can be evaluated using our simulation technique, then, as development progresses more and more details can be added to these models to more accurately capture the behavior of the system. At later stages the method can be used to derive reliability estimates of the system, but also to track down potentially hazardous execution scenarios.

In Chapter 1.1 we summarize the main contributions of this thesis, as well as the author's contribution to each paper. Chapter 1.2 provides some overall conclusions.

[1] One can therefore also argue that the real-time research community has wrong focus.

1.1 Results

This section summarizes the main contributions of each paper in this thesis. In addition we give information on the authors contribution to each paper.

Paper A

Markus Lindgren: *"Deriving Worst-Case Execution Time by Measurements."* Technical Report 00/26, Mälardalen Real-Time Research Centre, Mälardalen University, November 2000.

Summary: This paper introduces a new method for deriving the worst-case execution time of programs by performing measurements on the target hardware. As a basis for the method we use a well known result by McCabe [McC76] and others, namely, the observation that there is a set of independent program paths from which all paths within a program can be expressed. The core of the method is to derive linear equations describing the execution time for executed paths as the sum of the visited independent paths. This procedure is repeated for a set of input data to obtain a system of equations. The solution to this equation system gives execution times for the individual sub-paths.

The benefits of this approach are that: 1) measurements can be performed on the *whole* program; there is no need to perform measurements on isolated parts of the program, 2) the target hardware can be used for the measurements, which eliminates the tedious and error prone work of creating an accurate timing model of the target system. Both these issues are becoming increasingly important since the timing of many processor instructions are becoming context dependent. For instance, to derive the execution time for a basic block on a pipelined processor, it is important to know which basic block is executed before and after the block. Therefore measurements can no longer be performed on isolated parts of the program, the *whole* program must always be taken into account if we need accurate results. Our method avoids the need for exhaustive measurements of program paths by measuring only the independent sub-paths, which often are order of magnitudes fewer than the total number of paths.

On my effort: This paper is an extension of [LHT00] which is co-authored by Hans Hansson and Henrik Thane. Some errors in [LHT00] have been corrected and an experimental evaluation of the method added. This includes:

- the implementation of a program that instruments assembler code, such that it can be observed how many times each sub-path is executed for a particular input. The program is called META in the paper.

- the implementation of a program that can take these observations, derive an equation system and eventually solve the equation system when there are sufficiently many observations. This program is named SOFOP.

- adaption and bug fixes on the μSYSA processor which we use for our measurements.

- the implementation of an assembler, SDKASM, for the μSYSA processor. About half of it is written by me and half by Mohammed El Shobaki.

I am the main author of the original paper [LHT00]. Hans Hansson and Henrik Thane have participated in discussions on this work and provided feedback and comments on the paper. Henrik contributed to the introduction and the section on measuring execution time and Hans to the extension on pipeline analysis.

Paper B

Markus Lindgren, Hans Hansson, Christer Norström and Sasikumar Punnekkat. *"Deriving Reliability Estimates of Distributed Real-Time Systems by Simulation."* In Proc. of the 7^{th} International Conference on Real-Time Computing Systems and Applications; RTCSA'00, December 2000.

Summary: Real-time scheduling techniques has not gained widespread use in industry yet. One of the reasons for this is that the current methods seldom matches how systems are actually implemented, nor do they cope well with the hardware that is used. We should, of course, mention that there are areas in which real-time scheduling is used successfully, but also that there is a large class of applications for which it is not directly applicable, mainly those with soft real-time requirements. It is also important to note that the hard real-time requirements which exist in schedulability models do not exist in the real world, hence, these models often deal with "artificial" requirements.

In this paper we propose a novel simulation technique for this class of applications. From simulations of the system we observe when errors occur and then we make statistical predictions on the overall reliability of the system. The major benefit of this, is that the analysis can be performed on all kinds of systems. The drawback is that we use statistics, and hence, cannot provide strict guarantees about the timing behavior of the application (except in special cases). However, industrial applications often require a reliability estimate, rather than timeliness guarantees.

On my effort: I am the main author of this paper. I have also designed and implemented the simulator, which is used in the case-study.

The paper is based on ideas presented in [HNP00]. These ideas are extended, refined and made more concrete by incorporating them into a simulator. We also report additional experiments that further demonstrates the applicabilty of the method.

Hans, Christer and Sasi have been active in many discussions on the paper, and have been proof reading the paper and providing valuable feedback.

Paper C

Markus Lindgren and Christer Norström: *"Case-study: Using Simulation to Verify Real-Time Properties."* Technical Report 00/27, Mälardalen Real-Time Research Centre, Mälardalen University, November 2000.

Summary: This report continues the work on using simulation for reliability estimates. We apply the method to an industrial case-study, and identify future directions for our research.

The main conclusion is that methods for interpolation and extrapolation of execution time measurements observed on the target must be developed.

On my effort: I am the main author of the report and I performed the case-study. Christer Norström initiated the work, and he has also been providing valuable feedback throughout the process of writing the report.

1.2 Conclusion

This thesis has introduced two new methods that can be used in the development of real-time systems. We have also presented initial examples that support the hypothesis that these methods are useful in development of industrial real-time systems. However, further studies and developments are needed.

The work on measuring execution time has showed that it is possible to use measurements to derive the worst-case execution time of programs (under certain assumptions). Such results are important when performing timing analysis, e.g., schedulability analysis. The part on simulation has provided an easier way of analyzing complex distributed real-time systems, which has a much tighter coupling to the way systems are implemented, compared to existing methods for schdulability analysis. We believe that both of these contributions can be further developed into tools that are useful to industry.

1.3 Future Work

There are several directions in which the research can proceed, which we identify separately for each of the two parts of this thesis.

Using Measurements to Derive the Worst-Case Execution-Time

Strong limiting factors of our current technique are that it assumes that:

- loops have single entry and exit points and that backward edges are only allowed to loop entries.
- basic blocks have constant execution time, with the extension that pipelining effects between adjacent blocks can be handled.

It should be investigated if the approach can allow multiple entries and exits to loops, since it is common that these are generated by compilers. Multiple entries/exits create new sub-paths within loops, which can possibly be handled with the current method (although this has to be investigated).

Varying execution times of basic blocks results in an over-determined equation system, if sufficiently many measurements can be performed. Such equation systems can be solved by optimization techniques, such as the least square method. If these solutions are to be useful in real-time system analysis, confidence intervals should be computed for the derived values. These values can then be used to derive worst-case execution time estimates for programs, with a certain confidence value.

Furthermore, possible interfacing of our tools to symbolic execution tools for automatic removal of infeasible paths should also be studied. Finally, the method should be applied on industrial sized systems for evaluations and identification of additional issues.

Using Simulation to Verify Real-Time Properties

For the success of this method it is imperative that new methods are developed for the interpolation and extrapolation of execution times obtained from target system measurements. This will allow more execution scenarios to be captured during simulation, which increases the chance of revealing potential problems in the system. Furthermore, it is also important to develop a sound method for estimating the coverage in relation to the simulation time. This would, for instance, make it possible to determine if a certain simulation time is sufficient to with an acceptable high probability cover all relevant task phasings.

Case-studies are also important for further evaluation and tuning of the methodology.

Bibliography

[HL99] Hans Hansson and Markus Lindgren. Design and Scheduling of Shared Displays for Embedded Systems. In *Proc. of SNART'99*. Linköping University, 1999.

[HNP00] Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Hard Real-Time in a Soft World. Technical report, MRTC, May 2000.

[IL00] Damir Isović and Markus Lindgren. Real-Time Components. Technical report, MRTC, Sweden, March 2000.

[ILC00] Damir Isović, Markus Lindgren, and Ivica Crnkovic. System Development with Real-Time Components. In *Proc. of ECOOP2000 Workshop 22 - Pervasive Components*. Springer Verlag, 2000.

[LHNP00] Markus Lindgren, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Deriving Reliability Estimates of Distributed Real-Time Systems. In *Proc. 7^{th} Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 2000.

[LHT00] Markus Lindgren, Hans Hansson, and Henrik Thane. Using Measurements to Derive the Worst-Case Execution Time. In *Proc. 7^{th} Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 2000.

[Lin00] Markus Lindgren. Deriving Worst-Case Execution Time by Measurements. Technical Report 00/26, MRTC, Sweden, December 2000.

[LN00] Markus Lindgren and Christer Nortström. Case-study: Using Simulation to Verify Real-Time Properties. Technical Report 00/27, MRTC, Sweden, December 2000.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

Deriving Worst-Case Execution Time by Measurements

Abstract[a]

Execution time analysis is used in the development of real-time and embedded systems to derive the timing estimates required for schedulability and other timing analyses. Traditionally, execution time analysis is partitioned into high-level analysis, where the program flow is analyzed, and low-level analysis, where the execution times of program fragments (e.g. basic blocks) on a particular target architecture is analyzed. The execution time of the analyzed program is then obtained by combining flow information (such as number of iterations in loops) with the low-level timing information.

This paper proposes a method for low-level timing analysis based on measurements of execution times of programs executing on the actual target architecture. The essence of the method is to derive a system of linear equations from a limited number of timing measurements of an instrumented version of the considered program. The solution to these equations give execution times for program sub-paths, from which execution time measures for the entire program can be derived. The main advantage with this approach is that architectural modeling is not needed, hence the risk of a discrepancy between model and real system is avoided, as is the often tedious (and error prone) task of constructing the model. Also, compared to the non-exhaustive measurements performed in industry today, our approach is more structured and gives complete coverage in terms of the program paths considered.

We present our method in the context of a simple, but reasonably realistic, processor model and show how it can be extended to architectures with pipelines. Moreover, neither processors with caches nor super-scalar processors are studied. In this initial study we assume well-structured programs.

The applicability of the method is demonstrated by developing a set of associated prototype tools, and using them to measure the execution time of a simple program executing on an experimental processor architecture.

[a]This paper is a revised and extended version of [LHT00].

2.1 Introduction

Worst-case execution time (WCET) analysis or WCET estimation is crucial for successful analysis and implementation of hard real-time systems. The goal of WCET analysis is to generate a *safe* (i.e. no underestimation) and *tight* (i.e. small overestimation) of the worst case execution of a program. When performing WCET analysis it is assumed that the program execution is uninterrupted and that there are no interfering background activities, such as DMA or refresh of DRAM. Timing interference caused by such resource contention should be handled by subsequent analysis, typically by the schedulability analysis.

The process of analytically generating a WCET estimate from a program can be divided in *high-level analysis* (also referred to as program flow analysis), *low-level analysis*, and *calculation*. The task of the high-level analysis is to determine the possible paths through the program. The result is information about which functions get called, how many times loop iterate, if there are dependencies between different if-statements, etc. In the low-level analysis, the execution time of program fragments (typically basic blocks) on a particular target system is estimated. This analysis can be partitioned into *global low-level analysis*, in which the execution time effects of machine features that reach across the entire program are considered (e.g., cache effects), and *local low-level analysis*, which handles machine timing effects that depend on a single instruction and its immediate neighbors. Examples of such effects are pipeline overlap and memory access speed. Finally, in the calculation phase the WCET estimate is calculated, given the high-level and low-level analysis results. (For a more detailed and thorough overview of issues and work in the area of WCET analysis we refer to [PB00, EES+99, EES+00].)

This paper is focused on the low-level estimation of WCET. Such analysis is becoming increasingly difficult due to new processor features, such as pipelines, caches, and branch prediction units. Modeling of processors' timing behavior has been the way research has been approaching this problem, but this may become impossible in the future, since processor manufacturers are not always releasing information about the internal structure of processors, making it hard to create accurate models. Another difficulty, which afflicts all kinds of modeling, is how to determine if a model is correct or not. Using an incorrect model will, of course, invalidate any analysis based on it. The ever-increasing flora of processors is yet another problem, since each new processor family requires its own timing model.

In contrast, industry has relied on *measurements* of execution times when designing real-time software. The current practice in industry is however a trial and error process where no guarantees can be given, since exhaustive testing/measuring seldom can be performed [PF99, PN98, PB99].

The major problem facing measurements of WCET is the time required for performing the measurements. Consider for example a program with a single 32-bit integer as input. To be sure to find the WCET we would need to measure the execution time of the program for all possible inputs, i.e., make 2^{32} measurements (assuming that the program does not rely on internal state). Unless the execution times of programs are extremely short this process would take years to perform. The number of required measurements depend both on the program being measured and the hardware itself; for some architectures it is sufficient to test all paths through the program (although this is also time-wise intractable for nontrivial programs). The question is then if the number of measurements can be reduced, while still being able to produce safe timing estimates.

In for example [PK89], it has been proposed that the program should be divided into smaller parts, e.g., basic blocks [ASU86], which are then measured in isolation. This is a reasonable approach for some processor architectures, but not for all, since the timing of many processor features depend on the history of execution, i.e., the path and variable bindings prior to reaching an instruction. The history is especially important in analyzing pipelines, locality for caches, and variable bindings for certain types of instructions, e.g., division. Hence, timing behavior is often context dependent. Special care has to be taken when measuring execution time on isolated parts of software.

Fortunately enough from a WCET analysis perspective, the majority of processors used in embedded applications are simple 4-, 8-, and 16-bit processors for which WCET analysis is a simple matter of adding up execution cycles for individual instructions. This corresponds well to the *sum of basic blocks* execution time model, used in e.g. [LMW95]. This model assumes that the execution time of a sequence of basic blocks is the sum of the individual execution times of the included basic blocks. Furthermore, the more advanced processors used in embedded and safety-critical applications, such as the popular ARM micro-controller family [ARM], have a single simple pipeline, and very few have caches. In fact, for critical applications the silicon area that could be used for on-chip caches is usually traded for ROM/RAM to which critical code and data can be statically allocated. Hence, methods for WCET estimations of relatively simple processors has a potential of satisfying the execution time analysis needs for a majority of embedded and safety-critical applications.

Contribution

This paper presents a novel solution to the low-level analysis problem for such relatively simple architectures, i.e., those with no global low-level effects and (possibly) with only a simple pipeline.

The proposed method derives the execution time of program sub-paths by measuring the execution time of the *whole* program. Faced with intractable quantities of test cases for exhaustive execution time measurements, the method presented in this paper only requires a limited number of measurements of the entire program in order to derive the execution times of the program sub-paths.

By executing and measuring the execution time of the entire program, instead of just measuring program fragments in isolation as done by [PK89], we avoid the context sensitivity of certain instructions. The method provides timing information of program sub-paths to be combined with high-level WCET analysis providing information about loop iterations and feasible paths. Hence, the method assumes a context with high-level analysis and calculation methods, e.g., as proposed in [EES+99, EES+00].

Paper Outline

In Section 2.2 we give an overview of our method. Section 2.3 describes which timing information can be derived from programs, and how this can be performed. In Section 2.4 we show how the method can be extended to simple pipelines. Section 2.5 describes the experimental setup we have used to verify the method. Section 2.6 uses an example to illustrate the concepts and ideas. Evaluation of the method and open issues are discussed in section 2.7. In Section 2.8 related work is presented. Finally, Section 2.9 concludes the paper.

2.2 Method Outline

The method for timing program sub-paths presented in this paper assumes input in the form of a control flow graph (CFG) of the analyzed program, an execution time analysis (WCET) module calculating the final WCET estimate based on both flow information, such as maximum number of iterations of loops, and the timing of program sub-paths provided by our method. A typical context handling this is illustrated in Figure 2.1. It should be noted that we assume a direct correspondence between the CFG and the assembly code. Hence, there may, or may not, be a correspondence with the source code, depending on the transformations/optimizations performed by the compiler. How optimizations influence execution time analysis and the program flow is treated in [Eng97], which, however, is outside the scope of this paper.

The basic idea of our approach is to describe the entire control flow of a program by a set of linearly independent paths, as used in the classical complexity measure by McCabe [McC76]. From a set of linearly independent paths it is possible to express all paths of the program as linear combinations of the independent paths.

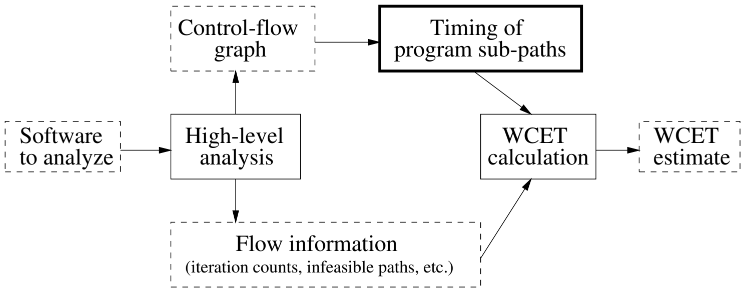

Figure 2.1: The context of our method for timing program sub-paths

A set of independent paths is for a well-structured program just a fraction of the number of possible execution paths through the program. This allows us to reduce the number of measurements to a set of linearly independent paths.

In essence the method is performed in five steps:

1. Identify a set of linearly independent paths. We will here assume that a separate flow analysis module has done this.
2. Instrument the target software in order to be able to determine which independent paths are visited during execution of the program.
3. Run and measure the execution time of the target software, while monitoring the sub-paths that have been visited. This has to be repeated with different input data until a sufficient number of paths has been covered (see step 4 and 5).
4. Set up an equation system of linearly independent equations representing the visited sub-paths as unknowns and the measured execution times of the target program for these paths.
5. Solve the equations. The solution provides execution time estimates that can be fed into the calculation of the final WCET value.

As an illustration, consider the simple CFG in Figure 2.2. The independent paths in this program can be chosen as *abce*, *abde* and *abcebce*; there are several possible sets of independent paths and anyone of these suffices. We can also express all paths of the program with the regular expression $a(b(c \vee d)e)^+$.

Which timing attributes should we now derive? For Figure 2.2, this could for example be what is the execution time (ET) for the left path through the loop in Figure 2.2, what is the ET for basic block a, or what is the ET for basic block $b + d$? Typically, to be able to calculate the WCET of a program we need the

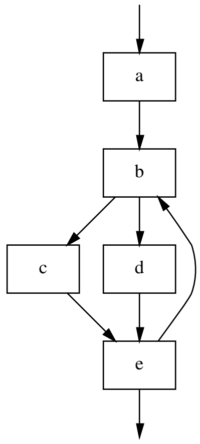

Figure 2.2: A control-flow graph.

execution times for the different paths through the program, expressed either as execution times of individual basic blocks or combinations of basic blocks.

Using our approach only timing for some of the programs sub-paths can be derived; namely a, bce and bde. To achieve this we must be able to observe how many times each of these sub-paths are executed during a measurement, e.g., that the sub-path bce was executed nine times for a particular input.

Since we assume the sum-of-basic blocks model [OS97], the ET for a program equals to the sum of the ETs for the basic blocks that are executed. This means that if we run, for instance, 5 tests with different input data we may end up with the following 5 equations defining the execution time of the different tests (where $ET(x)$ denotes the execution time of sub-path x):

$$
\begin{array}{lrcl}
(1) & ET(a) + ET(bce) & = & 40 \\
(2) & ET(a) + 2 \cdot ET(bce) + 2 \cdot ET(bde) & = & 121 \\
(3) & ET(a) + 2 \cdot ET(bce) + 3 \cdot ET(bde) & = & 143 \\
(4) & ET(a) + 3 \cdot ET(bce) + 2 \cdot ET(bde) & = & 158 \\
(5) & ET(a) + 3 \cdot ET(bce) + 12 \cdot ET(bde) & = & 378
\end{array}
$$

To solve such a set of linear equations with three unknowns, we need three linearly independent equations. Fortunately, we already have that, since the above equation system can be reduced to:

$$
\begin{array}{lrcl}
(6) & ET(a) + ET(bce) & = & 40 \\
(7) & ET(bce) + 2 \cdot ET(bde) & = & 81 \\
(8) & ET(bde) & = & 22
\end{array}
$$

Solving this system of equations gives us $ET(a) = 3$, $ET(bce) = 37$ and $ET(bde) = 22$. Note that we have assumed that a taken jump requires as much execution time as a non-taken jump. In Section 2.4 we will describe how such effects can be handled.

Now, assuming that the high-level analysis has concluded that the maximum number of iterations of the loop is 30 and that the sub-path *bce* in the loop is executed at most three times, we can conclude that the WCET is $ET(a) + 3 \cdot ET(bce) + 27 \cdot ET(bde) = 708$. A value which was obtained by a limited number of tests, compared to performing exhaustive testing, which in most realistic cases is intractable. If we were to exhaustively test this program up to 60 tests would be required, which is to be compared to the three tests required by our method. Should the maximum number of iterations be 100, then up to 200 tests would be required, while our method still only requires (a minimum of) 3 tests.

2.2.1 Limitations of the Approach

It should be noted that this solution only has general validity if the execution times of paths are input data independent. This is true for the majority of instructions for most simple architectures, but does not always hold for more advanced instructions, such as division (and in some cases multiplication). We plan to address this problem in future work.

Furthermore, the method is not applicable to systems with caches. The class of applications we target either do not include caches, or always configure caches to act as fast RAM/ROM for parts of the address space.

In Section 2.4 we present an extension capable of handling simple pipelines. This extension is, however, not applicable for super scalar processors, which on the other hand are rarely used in safety critical systems.

2.3 Path Analysis

This section focuses on how the program can be instrumented in order to retrieve information on which sub-paths have been executed, i.e., how the coefficients and variables in the equation system are found.

The idea of our method is to run the program, measure the execution time, and observe the number of times each sub-path is executed. To retrieve information about the sub-paths that are executed there are two approaches that can be used:

Instrumentation Instrument the code in such a way that information on which sub-paths are executed can be obtained (details in Section 2.3.2). This is performed by inserting extra code which increments an associated counter for each sub-path. A disadvantage with this approach is that these probes increase the code size and execution time. The probe effects can, however, be eliminated by running the test in two phases:

1. Instrument the program (as above) and execute it to observe which sub-paths are executed.
2. Do the measurement again with the same input data, but without the probes, and measure the execution time. This method requires twice as much time, but alleviates the probe-effect problem altogether.

The input data used for the program can be generated randomly, by some heuristic method or taken from logs of the program executing in its target environment.

Control-flow analysis By using control-flow analysis we can derive input data for which the execution path is known. This information can be derived by, e.g., symbolic execution [CI95]. A disadvantage with this approach is that it may require considerable amounts of computing resources. However, this method may be faster than other approaches since it can effectively find "good" input data.

2.3.1 Control Flow Graph Reduction

To simplify the analysis for instrumentation of the code we first transform the CFG into several smaller CFGs. For each inner loop we create a CFG containing only those blocks that belong to the inner loop, and replace the loop in the original CFG with an edge from the predecessor to the successor of the loop. This can cause more inner loops to be created in the original CFG. We repeat the procedure of reducing inner loops until none remain. For example, in Figure 2.3 we have reduced the CFG to the left in the figure to three smaller CFGs. First, we identify blocks $B3$–$B6$ as an inner loop, and hence, we create a new CFG containing only those blocks. The blocks are then replaced by a single edge from block $B2$ to $B7$ in the original CFG. This causes a new inner loop to be constructed, containing the blocks $B1$, $B2$, $B8$ and $B7$. We create a new CFG for this loop as well and replace it with a single edge in the original CFG, leaving only block $B9$ in it.

Loops form the basis of the independent paths and can be analyzed separately. This can be seen by expressing all paths in the program as regular expressions, and matching these with the sum of the regular expression for the constructed CFGs.

Now, the execution time of each sub-path in the resulting CFGs become an unknown in the equation system. For these transformations we assume that the CFG is well structured, i.e., each loop has a single entry and exit point, and backward

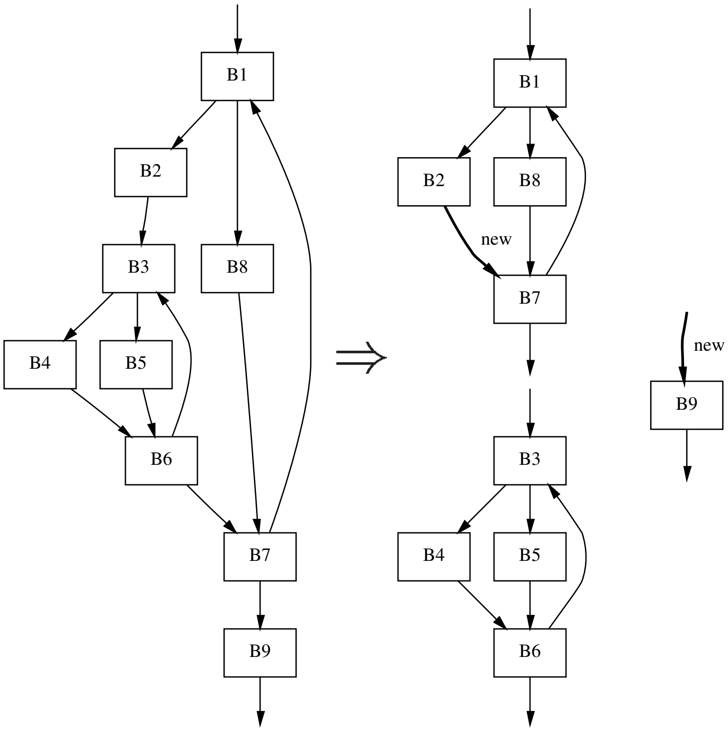

Figure 2.3: Transformation into several smaller CFGs without inner loops

edges are only allowed to loop back to entry points. Note that the transformations we perform are consistent with the entities needed in the calculation phase, which requires the entities to be consistent with those used by the high-level flow-analysis tools, e.g., we may combine two basic blocks that are executed in exactly the same loop iterations, but not blocks executed in different alternatives of a selection.

To aid the analysis even further we group blocks that are always executed equally many times for all input data, e.g., basic block $B3$ and $B6$ in Figure 2.3. This is done by exploring all paths in the CFGs and grouping those blocks that have exactly the same occurrences for all paths. The next section describes how this information is used to instrument the code.

2.3.2 Sub-Path Instrumentation

The objective of this section is to define an algorithm for instrumenting the code in such a way that it is possible to determine how many times a sub-path has been executed within a CFG. Consider as an example the CFG in Figure 2.4. How should the code be instrumented in order to be able to observe how many times

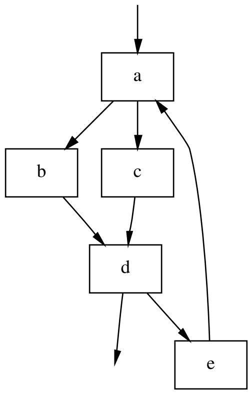

Figure 2.4: There can be two independent sets of sub-paths within a loop

each sub-path is executed? Firstly, we have to observe that within each loop it may become necessary to do two separate sub-path instrumentations: one from the start (block a) to the end (block d) and one from the end to the start, i.e., the sub-paths are independent between these two blocks.

Secondly, we have to find a way of distinguishing between different sub-paths. This is not obvious how to do since a single basic block does not always uniquely identify a sub-path, i.e., it is not always sufficient to only count how many times a basic block is visited. For example, in Figure 2.5 we are required to keep track of more than a single block to determine which path has been executed (we will return to this example later).

There are several different approaches that can be used to observe the sub-paths, they differ only in their implementation and memory requirements. We present three ways of instrumenting the code, and focus on the method that require the least memory of the three. Common among the approaches is that they all require that *path distinguishing* blocks have been identified, marked with bold boxes in Figure 2.5.

A path distinguishing block/edge is a block/edge between the split and join of a selection in a CFG that uniquely identifies a sub-path. For example, in Figure 2.5 block a is a split and the corresponding join is block d. Between these block there are two paths, and hence, there must be two path distinguishing blocks; namely b and c. The next split/join pair that needs to be investigated is d and i. Between these we find the blocks g, h and f to be distinguishing. Note that block e is not path distinguishing, since it occurs in both sub-path eg and eh.

The perhaps simplest approach to identify the executed path is to assign an identifier to each path distinguishing block, and let the block write its identifier to a global variable when it is executed. There need to be one global variable per

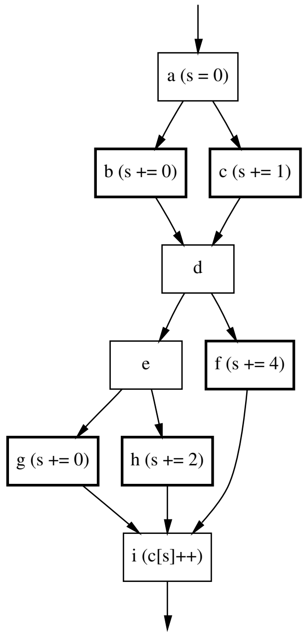

Figure 2.5: Observation of sub-paths within a CFG

split and join in the CFG (each variable is written to at most once for each path). The last block of each sub-path then checks these variables to see which combination of path distinguishing blocks have been executed, and registers this as an observation. Initially all associated global variables should be assigned an unused identifier.

A second alternative would be to let each block push its identifier to a tape/stack (one tape per CFG). This tape would then be sent from the target (where the measurement is taking place) to the host, where it is analyzed to determine which sub-paths have been executed. This requires that entire paths are recorded on the tape, which can require a very long tape (possibly infinite).

Both the above solutions could be useful, since measurements can be performed in two steps, as described in Section 2.3; one round where the visited sub-paths are discovered and one where the execution time is measured. However, for most realistic programs and systems they are likely to require too much memory, both in terms of code and data, since the number of basic blocks tends to be large. We now focus on a method which achieves the same thing as the two above, but with much lower memory requirements (reduced to a single global variable per CFG).

A more efficient solution

The aim of the third solution is to assign a number to each sub-path within the CFG, such that the sum of numbers assigned to executed sub-paths uniquely determines the executed path. During execution the sum is computed by a simple addition statement in each *path distinguishing* block (as indicated in Figure 2.5).

The intuition behind the method is to assign a number to each path distinguishing block/edge, such that the sum of the numbers for each path computes a unique number. To minimize memory requirements the path identifying sums should range from 0 to $n-1$, where n is the number of paths in the CFG. For the example (Figure 2.5) the numbers inside parenthesis will give such sums. Each path through the CFG computes a number, s, in the range 0–5 that is used to increment the counter, c, which is associated with the unknowns in the equation system for the particular CFG/loop. We now present an algorithm for assigning these numbers on a more general CFG.

The algorithm assumes that the CFG has been transformed as described in Section 2.3.1, and that "sequential" blocks have been grouped. Furthermore, it is assumed that all CFGs are well-structured. The algorithm is applied twice on each CFG separately; once between start and end block of the CFGs and once between end and start block. This captures both sets of independent sub-paths within a CFG (see Figure 2.4).

The input to the algorithm are:

- $\mathcal{B}$, the current group of blocks being analyzed.
- $\mathcal{C}$, the CFG.
- the set of grouped blocks in the CFG $\mathcal{G}$. The grouped blocks are stored in lists, which are sorted in the order the blocks appear in the CFG.
- $\mathcal{V}$ set of visited groups of blocks.
- *base* a number used to compute the annotation.
- *offset* a number used to compute the annotation, which is passed by reference (any changes to it is also reflected on the calling side).
- p the current path number.
- *level* the nest level for the current group of blocks.
- *id*, a unique identifier for the current CFG.

The algorithm returns the number of paths that exists between the start and end of the current group of blocks $\mathcal{B}$, and also the instrumented CFG. Please note that this algorithm does not work on all possible CFGs, but it still gives the general idea on how it can be handled. Currently, graphs that result from short-circuit logic cannot be handled.

```
annotatePaths(B, C, G, V, base, offset, p, level, id)
begin
  V = V ∪ B
  paths = 1
  b = getBlockBeforeNextSplitInGroup(nil, B, V)
  if l = 0 then
    annotateZero(head(B), id)
  endif

  if no block b then // path distinguishing block?
    if p > 0 then
      annotateAdd(head(B), id, p, base, offset)
    endif
  else
    loopTurn = 0
    index = p
    startBase = base
    startOffset = offset
    while block b exists do
      if loopTurn = 1 then
        base = base × p
        index = 0
        offset = 0
      elsif loopTurn > 1 then
        index = 0
      endif
      p = 0

      for both successors b1 and b2 of block b
        if inSameCFG(b, bx, C) then
          tmp = groupOf(bx, G)
          p = p + annotatePaths(tmp, C, G, V, base, offset, p + index, level + 1, id)
        endif
      endfor
      paths = paths × p
      loopTurn = loopTurn + 1
      b = getBlockBeforeNextSplitInGroup(b, B, V)
```

```
    endwhile
    // compute offset in old base
    offset = paths × base/startBase+ startOffset −1
  endif

  if l = 0 then
    annotateIncEquation(B, C, id)
  endif
  return paths
end
```

The function `getBlockBeforeNextSplitInGroup` finds the next block in the current group that has two successors, which have not already been visited. `annotateAdd` instruments the code such that the global variable belonging to the loop is added with the following number:

```
i = 0
if p > 0 then
  i = p
  if base > 1
    i = offset = offset +1
  endif
endif
// add with number i × base
```

To show how the instrumentation is computed we will now apply the algorithm to Figure 2.5. Firstly, we have to identify the different groups of blocks, which are $\mathcal{G} = \{\{a, d, i\}, \{b\}, \{c\}, \{e\}, \{g\}, \{h\}, \{f\}\}$. The initial call is annotatePaths($\{a, d, i\}$, $\mathcal{C}$, $\mathcal{G}$, {}, 1, 0, 0, 0, $myId$). Next, we find the block before the first selection in the group, which is block a (the result from `getBlockBeforeNextSplitInGroup`). Then we apply the algorithm to the successors of a; annotatePaths($\{b\}$, $\mathcal{C}$, $\mathcal{G}$, $\{\{a, d, i\}\}$, 1, 0, 0, 1, $myId$). For both block b and c we do not find any block before the next split, hence, we have identified these as path distinguishing blocks that need to be instrumented. Block b is instrumented with 0 and c with 1.

Now the first phase of the algorithm is complete, we have counted the number of paths between the first and second block of the first group in the CFG. Two paths were found, which also gives the new base (2). The block before the next split in the group is d. We then apply the algorithm to d's successors (e and f). First e, annotatePaths($\{e\}$, $\mathcal{C}$, $\mathcal{G}$, $\{\{a, d, i\}, \{b\}, \{c\}\}$, 2, 0, 0, 1, $myId$).

For e we again find that the block has two successors, which we have to apply annotatePaths to. The difference compared to the earlier case, is that block h

uses the base and the offset when computing its instrumentation ($base = 2$, $p = 1$, $offset = 0 \Rightarrow offset = 1$, $p = 2$).

When we carry on with block f, we have the following variable bindings: $base = 2$, $offset = 1$, $p = 2$. Block f is also a path distinguishing block that have to be instrumented. First we increment the offset and then compute it as $base \times offset$ ($= 4$). The last block in the first group (i) does not have any successors within the CFG which then terminates the function call.

Implementing `annotateAdd`, `annotateZero` and `annotateIncEquation`

Here we describe a technique that can be used to instrument blocks and edges in the CFG, i.e., how `annotateAdd` and `annotateZero` can be implemented.

Blocks are simple to instrument, since all that is needed is to insert code at the beginning of the block that does not change the functionality of the program. This usually requires that the status register, and other registers that are used during the instrumentation are saved, and then restored afterwards. For example, the following code can be used:

```
pushf   // push status flags on stack
push registers that will be used
perform operation, e.g., add cfg1sum,2
restore registers
popf    // restore status flags
```

Edges, however, require slightly more effort to handle. To be able to observe when an edge e is executed we must: 1) use a global variable $prev$, 2) assign 1 to $prev$ in the block preceding e, 3) assign 0 to all predecessor blocks of the block succeeding e (except the predecessor block of e), and lastly, 4) in the block succeeding e perform the appropriate operation when $prev$ contains the value 1.

The operation performed by `annotateAdd` is to add a number to a counter (global variable) which is associated with the CFG, and `annotateZero` assigns the value zero to the counter. `annotateIncEquation` increments the counter that is associated with the value found in the global variable for the CFG, i.e., this operation computes the coefficients in the equation system for the sub-paths.

The instrumentation requires some extra assembly code per basic block, and one global variable per edge that is to be observed. The overhead (memory and code) can be eliminated altogether by running the tests in two phases as described in Section 2.3.

We now have a method to identify the unknowns of a linear equation system. However, a solution can only be found if we can execute a sufficient number of independent paths, from which we obtain coefficients and execution time. How to handle situations when a sufficient number of independent paths cannot be executed is treated in the following section.

2.3.3 Infeasible Paths

The number of feasible paths in a program can limit the number of obtainable independent measurements. Should it be that the infeasible paths prevent us from solving the equations, all sub-paths that are infeasible must be removed from the equation system. In such cases the analysis algorithm will prompt the user to manually change the equation system, by for example selecting sub-parts of the feasible paths or the entire paths. It should be noted that such modifications may correspond to removing some paths that have not been proven to be infeasible. Hence, there is definite risk that the obtained WCET is unsafe, since one of the removed paths may correspond to the actual worst-case. On the other hand, if no manual modifications are needed we have complete coverage in terms of the paths considered.

Some real-time programs contain code that has a *single feasible path* (SFP). For example, filters can have a single path through a large set of basic blocks. This prevents our approach from finding sufficiently many independent measurements. However, it makes little sense to provide detailed timing information for parts of code that has the SFP property. If code that contain SFPs can be identified we can replace that part of the code with a single path. A method for determining if parts of a program have SFPs has been presented in [EY97].

A special case of infeasible paths are those containing dead code. Deciding whether a piece of code is dead is in the general case infeasible [ZAHM97], but several methods handling special cases have been proposed [Alt96, Gus00, CI95]. Paths/equations corresponding to dead code can safely be removed.

2.3.4 Measuring Execution Times

There are several possible approaches to measuring the execution time, each with its own advantages and disadvantages. Here we review some of the possibilities [CBG00, PN98, PF99].

Using Clock Cycle Counters

A *clock cycle counter* counts clock cycles on the bus or from the CPU core. The benefits are that this approach is exact, time can be measured with a fine granularity, and it is non-intrusive, i.e., it does not disturb the execution of software in any way.

The disadvantage is that special hardware is often required, at least during the measurement phase. Some commercial processors have these counters built into them, e.g., Intel Pentium. However, for a large part of the embedded systems market this is not a viable approach since most micro-controllers do not have support for such features. The alternative is to build or buy one and attach it to the bus, i.e., the clock cycle counter can be either internal or external to the processor. Since time is measured in clock cycles the clock frequency and its tolerance must be known, in order to be able to compute the time that has elapsed. This is, however, a minor problem.

Using Timers

Timers can be found in most systems. These can be implemented in two different ways: fully in hardware or a combination of software and a periodic interrupt to the processor. Real-time clocks (RTCs) are used to implement the hardware (only) solution, and is accessed in a similar way as clock cycle counters, with the difference of having a lower resolution. Below we describe the "software" approach.

In micro-controllers the periodic interrupt is typically triggered by a programmable division of the frequency of some clock driving (not necessarily the processors clock). A counter (CN) decrements the clock cycles from a programmable preset and generates a timer interrupt to the processor each time it reaches zero. A periodic tick can thus be generated, usually with a period time ranging from microseconds to seconds. We can thus measure the execution time of programs by counting these ticks. The granularity of the ticks can however be too coarse, which in most cases can be handled by also reading the CN. The execution time can thus be given a value in ticks + the fraction of a tick using the CN.

The disadvantage with this approach is that the timer interrupt will disturb the timing behavior of the program being measured (the interrupt routine requires some processor time). This is however in practice not a problem since the number of ticks says how many interrupts have occurred. If we know the execution time overhead of the interrupt service routine it is straightforward to subtract the accumulated overhead from the measured execution time.

The interference by the timer tick interrupt does nonetheless complicate matters when processors have pipelines and caches, since the periodic interrupt can prolong the execution time of programs due to flushed pipelines or cache misses, but also decrease the execution time (in the case where the interrupt routine brings in cache lines that are later used by the program; this is, however, not likely to occur for "real" programs).

The question is however what the effect of this last intricacy has on real systems, since most real-time operating systems are driven by a timer interrupt, and that the effects of the timer interrupt will thus occur during runtime.

In this paper we assume, for simplicity, that clock cycle counters are used, since this will keep the presentation reasonably simple.

2.4 Extensions for Pipelines

The method presented this far is appropriate for architectures where the execution time of basic-blocks are fixed and independent of the surrounding code. This is not the case for pipelined architectures. In this section we will show how our method can be extended to also handle architectures with simple pipelines.

In a pipelined architecture the execution of instructions is divided into several phases which for different instructions can be performed in parallel, e.g. while the operation of one instruction is being performed, the result of the preceding instruction may be stored and the operands of the succeeding instruction fetched. This parallelism and instruction overlap increases the processor speed. On the basic-block level, the execution of one basic block may overlap the execution of another, as illustrated in Figure 2.7. The effect of this overlap is a reduction in execution time, i.e., for the example in Figure 2.7, $WCET(a) + WCET(b) > WCET(a + b)$, which is a violation of our sum-of-basic-block assumption.

The way we avoid this problem is graphically illustrated in Figure 2.7. As can be seen, we make an artificial partitioning of the basic blocks so that the "artificial" basic blocks do not overlap. This is done by separating the overlapping portions of the original basic blocks into an interface basic block. In the example we combine the overlapping parts of a and b into the artificial block ab and rename the remaining parts of a and b to a' and b'. The outer edges of a and b will similarly be combined with preceding and succeeding basic blocks. The limitation of this approach is that it does not cover overlaps of multiple basic blocks. Our simple pipeline assumption means exactly this: we only allow overlap with immediate neighboring basic-blocks. This is, however, not a severe limitation, since more advanced pipelines are rare in current micro-controllers architectures (e.g. [ARM]).

To see the effect of our pipeline handling for the analysis of an entire program, consider the graph in Figure 2.6 which is derived from the CFG in Figure 2.2.

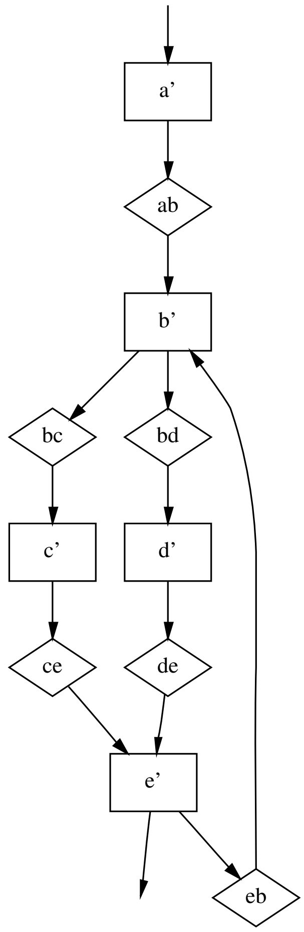

Figure 2.6: A CFG annotated with blocks for pipelining effects

The added blocks in diamond boxes in the figure denote the artificial blocks corresponding to the overlap of original blocks.

CFG-reduction

The obtained graph still satisfies our requirements on CFGs on which our program instrumentation algorithm and groupings algorithm can be applied (if we regard all nodes as basic blocks). Hence, we can use the algorithms to identify a solvable system of equations. Pipeline blocks are instrumented using the same technique as edges (as discussed in Section 2.3.2).

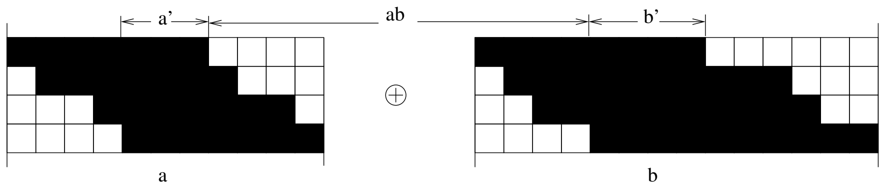

Figure 2.7: Concatenation of basic blocks executing in pipelined fashion

"Pipeline effects" in non-pipelined architectures

It should be noted that the execution time of a basic block may vary even for non-pipelined architectures, depending on which exit is taken, e.g., the execution time of conditional branch instructions is for most architectures dependent on whether the branch is taken or not. There may also be an influence from the preceding basic block, since different incoming paths to the block may lead to different status of prefetching and phasings of interleaved memory interfaces, leading to path dependent execution time of the basic block. Consequently, even for non-pipelined architectures pipeline type of effects between basic blocks are common. Our pipeline modeling will, however, accurately handle all such effects that are limited to only influencing the immediate neighboring basic blocks.

It should be noted that we do not make any explicit partitioning of the basic blocks as indicated by Figure 2.7; what we do is to add some unknowns representing the edges of the original graph to the equation system. These unknowns will capture the difference in execution time when executing a basic block in sequence with an other basic block.

2.5 Experimental Setup

This section describes the experimental setup we have used to verify our method. The target system is based on an experimental micro-processor architecture (μSYSA) used in the under-graduate course *System Architecture I* at Mälardalen University. We start by giving an overview of the components, and then we go into detail on individual components.

2.5.1 Overview

The objective of the experiments is to derive the worst-case execution time of program sub-paths of a program $\mathcal{P}$ executing on the target processor. The experimental setup we have used is illustrated in Figure 2.8. Below are brief descriptions of each component in the figure.

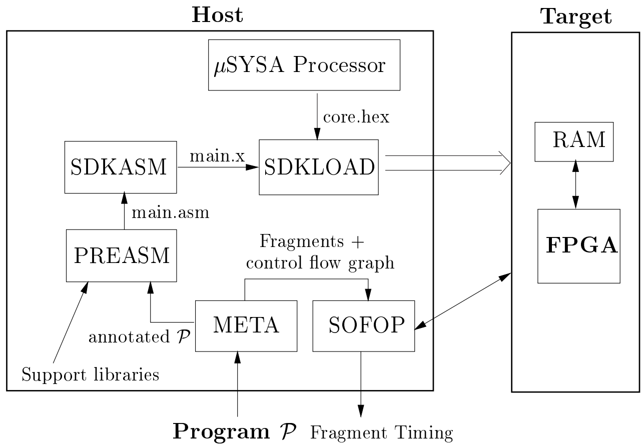

Figure 2.8: Components in the experimental setup

$\mathcal{P}$ is an assembly program for the target processor for which timing should be derived.

MEasurement and Timing Analysis (META) is a tool which groups basic blocks of $\mathcal{P}$ into fragments. In addition, it annotates the code for $\mathcal{P}$ in such a way that it is possible to extract information on the sub-paths within a loop that have executed for an input $\mathcal{I}$. META informs the SOFOP-tool how many sub-paths there are in the annotated program, i.e., the number of unknowns in the resulting equation system.

PREASM/SDKASM/SDKLOAD PREASM is a pre-stage which performs macro expansion (using the GNU C pre-processor) and merging of source files. This is just to aid the development of software for the target.

SDKASM is an assembler for the target processor, capable only of assembling *one* file at a time. (No linker is available.)

SDKLOAD is a program which transfers the result from SDKASM to the FPGA and RAM on the target. The data is transferred using a one-way parallel connection to the target.

core.hex is a file with the binary representation of the **μSYSA processor**. The processor and associated I/O units are written in VHDL. This file is downloaded to the FPGA for "execution".

Field Programmable Gate Array (FPGA) is the piece of hardware on which the μSYSA processor "executes". In essence any hardware design which can be implemented in VHDL can be downloaded to this chip, as long as it meets the capacity constraints of the chip. We use the Xilinx chip XC4062XLA, which has an capacity of 5472 logic cells ($\approx$ 40-130K gates).

SOFOP Generates random input $\mathcal{I}$ to the program $\mathcal{P}$. The input data is sent via a serial link to the target processor, which after execution responds with the total execution time for $\mathcal{P}$ with the input $\mathcal{I}$, together with information on executed sub-paths.

By repeating this procedure, SOFOP eventually collects sufficient information to define a solvable system of linear equations. The solution in terms of the timing for individual sub-paths is then calculated.

Support Libraries contain code for safe communication from host to target and vice verse, among other things.

We use a standard PC running Linux as a host in this setup.

2.5.2 μSYSA Processor

The μSYSA[1] processor is a non-pipelined 32-bit RISC machine. Its instruction set contains 32 instructions, which is a superset of the instructions in the popular Motorola PowerPC processor family. The processor executes at a clock rate of 10MHz and instructions usually require 11 clock cycles for execution. The execution time for instructions is input data independent, hence, the execution time of basic blocks are constant, i.e. the processor has the sum-of-basic-blocks property.

The internal structure of the processor is illustrated graphically in Figure 2.9. There are four main components in the design: 1) the μSYSA Core which is the core of the processor, 2) the MEMIF which is an interface to 256KB of external SRAM memory, 3) the UART is a standard serial line controller operating at 9600bps, which provides services for sending and receiving data, 4) the IRQ handler which decides when the core should be notified of interrupts, and 5) the TIMU which counts the clock cycles on the bus, which we use for measuring execution time. The next sections go into detail on some specific issues related to this processor.

Instruction Set

The basic idea behind Reduced Instruction Set Computers (RISCs) is to have few and simple instructions which can be executed rapidly. More complex instructions

[1] The development of the μSYSA processor is an obligatory assignment in a course for second year students at Mälardalen University.

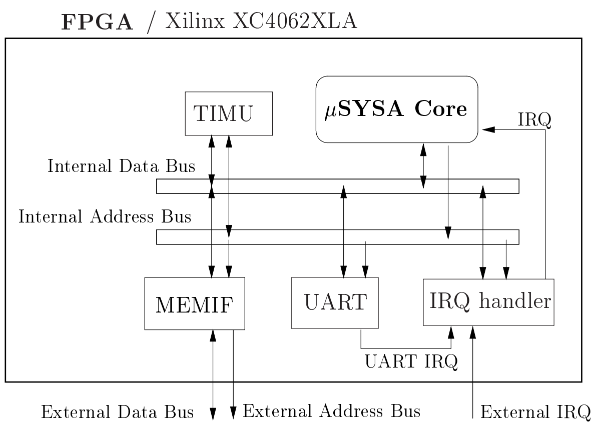

Figure 2.9: Internal structure of the μSYSA processor

are formed by combining sequences of "simple" instructions. The μSYSA processor builds on the same concept, but for a different reason, namely, making it simpler for the students to implement their processor. High performance is *not* a design criteria for this processor.

The processor has a register bank of eight registers, $r0$–$r7$, where $r0$ always holds the value 0. Arithmetic and logic instructions can only operate on registers and in some cases 16-bit immediate values. There are load and store instructions for fetching and storing data in memory (these are the only instructions which can access memory).

Instructions can be executed in either supervisory or user mode. In supervisory mode all instructions can be executed, and in user mode only non-privileged instructions can be executed. Supervisory mode is entered on interrupts and when calling the *trap*-instruction. Table 2.1 gives brief descriptions of the most common instructions. Worth noting is that there are no instructions for multiplication, division or floating point operations. These have to be written and put into support libraries if needed by the application. Also, all instructions occupy 32-bits of memory.

Besides the registers $r0$–$r7$ there are the *special purpose registers*: PC, LR, SRPC and SRSR. PC is the program counter that contains the address of the instruction which is to be executed, LR is the link register which is used to store the return address when doing sub-routine calls[2], SRPC (shadow register PC) is a supervisory register which stores the address of the next user mode instruction, and SRSR

[2]Usually this address is stored on the stack to handle calls to sub-routines from a sub-routine. On this processor the programmer is responsible for storing LR on the stack, to enable nesting of sub-routines.

Syntax	Semantics (C-stylish)
add rT,rA,rB	Addition $rT = rA + rB$
addi rT,rA,sIMM16	Addition with sign extended 16-bit immediate value $rT = rA + EXTS(sIMM16)$
addis rT,rA,sIMM16	Addition with 16 steps left shifted signed 16-bit value $rT = rA + (sIMM \ll 16)$
and rT,rA,rB	Logical AND $rT = rA \,\&\, rB$
andi rT,rA,uIMM16	Logical AND, see addi
andis rT,rA,uIMM16	Logical AND, see addis
b offset	Relative branch $PC = PC + offset$
bCC offset	Conditional branch $PC = PC + offset$ if condition CC true
bl offset	Branch linked $LR = PC$, $PC = PC + offset$
ldl rT,uIMM16(rS)	Load long word with displacement $rT = rS[uIMM16]$
or rT,rA,rB	Inclusive logical OR $rT = rA \mid rB$
rfs	Return from sub-routine $PC = LR$
rfe	Return from exception $PC = SRPC$, $SR = SRSR$
shl rT	Logical left shift $rT = rT \ll 1$
shr rT	Logical right shift $rT = rT \gg 1$
stl uIMM16(rT),rS	Store long word
trap IMM	Trap to supervisory mode
xor rT,rA,rB	Exclusive logical OR $rT = rA \wedge rB$

Table 2.1: Short descriptions of some of the instructions on the μSYSA processor

(shadow register SR) holds the contents of the user mode status register. These registers can explicitly be set by the instruction *mtspr* and retrieved by *mfspr*.

Executing Instructions

Here we briefly describe the different phases of executing an instruction on the μSYSA Core.

The processor has three sequential states for executing an instruction. In the first state *adrgen* the address of the next instruction is computed. This is done using the Arithmetic Logic Unit (ALU) in the μSYSA Core. The second state is *fetch* which fetches the next instruction from memory. Finally, in state *decode* the instruction is decoded and executed.

An average instruction requires 11 clock cycles to execute, and of those are 9 required for memory accesses. This comparably long time is due to a narrow data bus, only capable of transferring 16 bits per access. Hence, all states except those accessing external memory require a single clock cycle.

Benefits and Disadvantages of the μSYSA processor

We decided to use this processor for the proof-of-concept experiments since:

- It has the sum-of-basic blocks property, which we also can verify at source code level, i.e. the timing model is known.
- Expertise and support is available at our department.
- The timing behavior of the processor can easily be changed.

The disadvantages are:

- Initially, there was no way to measure execution time on the processor.
- The processor is not completely verified; there may be bugs, which will require some time to fix.
- There is no compiler available for the processor, hence, all code has to be written in assembly language.
- Initially, pipeline analysis cannot be performed on the processor, since all instructions has constant execution time.

Luckily, two of these problems, indicated by "initially" above, can quite easily be removed.

To be able to perform exact timing measurements on this processor we have added a memory mapped clock cycle counter to the bus, indicated by the TIMU component in Figure 2.9. We have also changed the timing behavior of conditional branch instructions, in such a way that a taken jump requires one extra clock cycle. This latter change implies that our pipeline analysis is required.

2.5.3 META

META is responsible for instrumenting the program which is to be measured, and to check that the program meets all the assumptions of the algorithm, e.g., that all loops are well-structured, and determine which the sub-paths are.

To do this META, must parse the assembly code and build a control-flow graph of the program; keeping track of where loops start and end, and also which basic blocks that belong to loops. After all checks, META groups blocks according to our algorithm, and lastly it emits the instrumented code.

2.5.4 Host/Target Communication

To provide reliable communication between host and target we decided to develop a simple data communication protocol. The protocol was developed for two reasons. Firstly, the UART only has a one byte buffer for received data and no flow control at all, hence data will be lost if not processed fast enough. Secondly, the processor we are using can execute about one million instructions per second and the incoming data rate is about three thousand bytes per second. These two reasons were motivations enough for a simple protocol. (During development it was also confirmed that the processor sometimes could not consume data fast enough.)

For error detection we use an arithmetic checksum (sum of data modulo 256). Character-stuffing is used for frame detection, where the special character STX denotes frame start and ETX frame end. All data between start and end is stuffed to avoid "false" frame starts/ends in the data. The checksum is stored before ETX.

Below is the pseudo-code of the host side of the protocol:

```
seqNo = 0;
forever
  /* generate input data I to program P */
  sendFrame(seqNo, I);

  /* wait for response from target */
  timeout = 1 second;
  status = receiveFrame(recvSeqNo, recvData, timeout);
  if(status != TIMEOUT)
    if(recvSeqNo == seqNo) then
      process received data, i.e., solve equations
    else
      ignore // ignore all other data, retransmit a new frame instead
  }
  seqNo = seqNo + 1
end forever;
```

The pseudo-code for the target is:

```
forever
  receiveFrame(recvSeqNo, inputData);
  // execute program (P) and observe fragments invocations
  sendFrame(recvSeqNo, observations);
end forever
```

As can be seen from the above pseudo-code, the target is only acting as a "dumb" slave in this protocol, only responding when it receives any input.

2.5.5 SOFOP

SOFOP is the tool that generates random input to the program $\mathcal{P}$, exchanges data with the target, and solves the resulting equation system that is observed on the target.

The number of inputs and input data ranges that are to be generated are specified when SOFOP is invoked. For example, the command line `sofop 1..30 1..30` causes sofop to generate two random numbers in the range 1–30. In addition, SOFOP needs to be informed on how many unknowns there are in the equation system. This information is passed on from META via standard input, e.g., `sofop 1..30 < meta-result.txt`. The grammar for specifying inputs to SOFOP is: `numlist: numlist numrange` | `numrange`, where `numrange` is specified by `number..number`.

The generated input is passed as an array to the program being measured, i.e., `func(int *array)` in C, where the first number is generated according to the first input data specification of SOFOP, second input specification to second number etc.

SOFOP iteratively generates input to the program until sufficiently many independent equations have been observed, and the system can be solved. Finally, SOFOP uses Gauss elimination to solve the equation system.

2.6 Experimental Results

Since we are not aware of any comparable techniques, apart from traditional WCET analysis and random or exhaustive testing, we cannot do any comparative analysis with other work. Instead we present a small case study, which illustrates the usefulness and details of the approach.

For our experiment we have decided to run the prototype tools on a program that has previously been used to validate WCET research results. The program we have chosen is taken from [EG97]. The reason why we have not benchmarked more programs is the quite time consuming job of translating the C-code of the program to the μSYSA assembly language, i.e., the disadvantage of not having a compiler. The C-source for the program is available in Figure 2.10.

```
1  int complex(int a, int b)
2  {
3    while(a < 30) {
4      while(b < a) {
5        if(b > 5)
6          b = b * 3;
7        else
8          b = b + 2;
9        if(b >= 10 && b <= 12)
10         a = a + 10;
11       else
12         a = a + 1;
13     }
14     a = a + 2;
15     b = b - 10;
16   }
17   return 1;
18 }
```

Figure 2.10: Example program

Our tools aren't currently capable of handling infeasible paths. Since the condition on line 9 is always false, we have therefore removed lines 9–11. Also, we have manually translated this program to μSYSA assembler.

To instrument the code and to determine how many sub-paths that need to be solved we first run META on the program. Next, we combine it with the communication routines for the target etc. SOFOP is then run on the host to generate random input data to the program, register observations from the host, and finally, solving the resulting equation systems.

After seven test-cases the following four independent equations were acquired:

p_1	p_2	p_3	p_4	Time [Clock cycles]
6	2	0	1	7292
0	0	0	1	1194
5	2	0	1	6491
1	2	3	1	5969

where p_1 denotes the sub-path that executes lines 4, 14 and 15, p_2 lines 5, 6 and 12, p_3 lines 5, 8 and 12, and p_4 lines 3 and 17, which are the sub-paths identified by META. Now, solving this system gives $p_1 = 801$, $p_2 = 646$, $p_3 = 894$ and $p_4 = 1194$, which is the desired output from our method. These values were also verified during simulation of the VHDL-code for the μSYSA Core.

Independent Equations

Above it has been shown that the equations can be solved for the example program. However, we did not give any information about how a system designer/engineer actually would obtain sufficiently many equations from a program.

This would typically be carried out in parallel with functional testing of the program, registering executing time and equations while performing the tests. The equations would then be reduced (by tools) to a set of linearly independent equations, as done by SOFOP.

The missing equations are obtained by 1) identifying them, which is done by inspecting the current equations, 2) deciding upon sub-path counts that should be obtained and 3) finding input data (see Section 2.3) that executes "along" the decided values.

For the example program this could easily be done manually, since there are so few sub-paths. For larger programs this step should preferably be handled by a tool as well.

Calculation

Assuming that we from high-level analysis know that both the inner and outer loop will execute at most 10 times in the program in Figure 2.10, a *safe* estimate of the programs WCET can be calculated as follows:

$$\text{WCET(P)} = \quad p_4 + 10 \times (p_1 + 10 \times \max(p_2, p_3)) = 98604$$

Industrial Practice

The industrial practice used in finding the WCET of a program is to subject it to different loads (input data) and measure the execution time for a period of time. This process lasts until someone decides that enough time and resources have been spent. Program analysis is rarely used to find input data that may be good candidates for finding the WCET. Instead, random tests are typically used.

Consider as an example the CFG in Figure 2.11, which has 12 independent sub-paths. The theoretical upper limit on the number of paths for this program is (if backward edges can be traversed at most 10 times each):

Inner loop: $p_i = 2 + 2^2 + 2^3 + \ldots + 2^{11} = 4094$
Outer loop: $p_o = p_i + p_i^2 + p_i^3 + \ldots + p_i^{11} + 8 + 8^2 + \ldots + 8^{11} \approx 5.42 \cdot 10^{39}$
Program paths $= p_o$

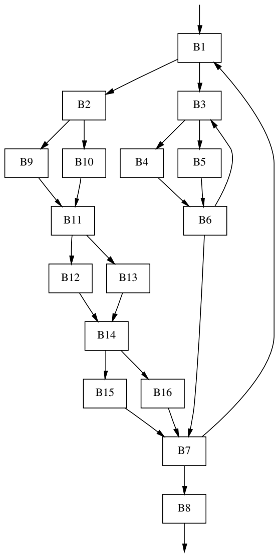

Figure 2.11: A relatively small CFG with many possible paths

If we assume that 10^{-6} of these paths are feasible and that a measurement/test on average requires 1ms, the time for performing exhaustive testing becomes $1.7 \cdot 10^{23}$ years (compare to the age of the universe which is beleived to be $12 \cdot 10^{9}$!). The time required for finding input data that executes along these paths must also be added (typically done by partitioning the input data into equivalence classes). Clearly, this is not a viable approach.

Instead random testing is performed for, say, a week, and 30% is added to the measured worst-case. During a weeks testing $6 \cdot 10^{8}$ cases can be covered; an extremely small part of the total state space of $5.42 \cdot 10^{33}$ paths. Hence, no guarantee can be given that the measured worst-case is *safe* or *tight*. If we assume that 1000 measurements are required to observe and solve the sub-paths using our approach, we still reduce the state space by several magnitudes (even though 1000 measurements is probably more than what would be needed).

2.7 Evaluation

Here we discuss how well this approach would work on "real" programs and how more advanced processors can be coped with. Doing this we also discuss pros and cons of the method, as well as open issues that require further studies.

2.7.1 Program Constructs

It has been concluded in [Eng99] that embedded programs do contain recursive functions calls, function pointers, and gotos (from machine generated code). These constructs have been ignored by some research groups with the motivation that they are hardly used. This is, however, not true, as [Eng99] concludes. Therefore, for a method to become successful it must be capable of handling these program constructs.

Function calls do exist in all realistic programs, and all realistic WCET tools must be able to handle them. Our method can handle function calls by instrumenting each function separately, and combining the two resulting lists of unknowns into a single list. During a measurement we only need to record the visited sub-paths and the total execution time (just as before). The only difference with having function calls or not is the number of unknowns.

Infeasible paths can become an even bigger problem when function calls are used in the code, due to data dependencies between the function and the callee. We will return to this issue in Section 2.7.2.

Using our approach we need not differentiate between recursive function calls and "ordinary" function calls, since they are handled in exactly the same way, i.e., no extra or special treatment is needed for recursive functions.

The use of gotos only changes the CFG, and hence, which sub-paths that need to be observed. As long as gotos do not change the loop assumption — single entry, single exit, and only backward edges to the loop entry block — this will not be a problem.

2.7.2 Infeasible Paths

As discussed earlier, infeasible paths poses a problem for our method. This problem becomes even more prominent when function calls are made.

Symbolic execution [CI95] could be combined with our method to automatically remove infeasible paths from the equation system. Typically, a large set of test-cases would first be executed to get an initial set of observations from the target. This set could then be analyzed to see which sub-paths that are missing. Backward analysis (using symbolic execution) would then be used to symbolically execute the program backwards, such that the desired path is executed. When the analysis reaches the program start, constraints on the input data has been found. An infeasible path has been found if the constraints cannot be fulfilled.

2.7.3 Hardware Architectures

In this paper we assume the sum-of-basic-block model extended to also handle overlaps/pipelining effects between adjacent basic blocks. This quite drastically limits the kind of target processors on which the method can be applied.

There are many building blocks in current contemporary processors that violates our sum-of-basic-block model, e.g., branch prediction units, caches and even memories with different access times (not only caches), certain types of instructions; multiplication and division are among those. So there are many things that make the execution time of basic blocks more or less unpredictable, and it is likely that there will be more of such architectural features in the future. It is also likely that these hardware features will be used more frequently in embedded systems, mainly because the price/performance ratio of such hardware is better than more "conservative" solutions.

Varying execution times of basic blocks results in an over-determined equation system, if sufficiently many measurements can be performed. Such equation systems can be solved by optimization techniques, such as the least square method, to find solutions to the equations. If these solutions are to be useful in real-time system analysis, it is also needed that a confidence interval is computed for the derived values so that estimations of the worst-case execution time can be obtained. This will require that more measurements are performed on the target, so that an over-determined equation system is obtained. It should, however, be borne in mind that such a method can only provide estimations derived from the *observations* that have been made, i.e., it cannot be guaranteed that the worst-case has been found.

2.7.4 Size of Instrumentations

As discussed in Section 2.3.2, the size of the instrumentation code and data must be kept within the memory capabilities of the target. For the μSYSA processor the code for instrumentations become rather large due to the primitive instruction set, on other processors the instrumentation code can be significantly smaller. Currently, we do not consider the size of instrumentations to be a problem with the solution described in Section 2.3.2.

2.7.5 Relaxing the Assumptions

The most limiting assumption we have made in this paper are the use of the sum-of-basic-blocks model, and that loops have a single entry and exit point. A method for coping with the first issue has been briefly discussed in Section 2.7.3. Here we discuss the single entry and exit issue.

Our proposed method needs to be able to observe which sub-paths that are executed in order to set up an equation system. How long or short these sub-paths are does not matter as long as the resulting equation system can be solved. Multiple exists from a loop can possibly be allowed, but this will require that the algorithm for instrumentations is improved. Instead of just applying it between two pairs of start and end blocks for a loop, all exits must be analyzed.

Multiple entries to loops can be constructed by optimizing compilers and sometimes also from machine generated code. This problem is similar to multiple exits, which also requires further studies.

2.8 Related Work

Not many have studied the problem of finding the execution time of parts of programs, in cases when the timing model of the processor is not known. Here we review some of the related work.

In early papers on execution time analysis [PK89] it was assumed that timing information for basic blocks were known, or that it could be obtained by 1) summing up the execution time for the assembler instructions in basic blocks or 2) measure the execution time for the blocks in isolation (without its context). These values were then used in, for example, the timing schema approach [PK89], to derive the WCET for entire programs.

In [PF99] a method for measuring the worst-case execution time of programs is presented. The method relies on the actual hardware for measurements, and tackles the problem of too many paths by dividing the program into so called measurement blocks. Each measurement block is instrumented with a function call that stores the current time (using the time stamp function available in Pentium II). Between measurement blocks, caches and pipelines are flushed, and in order for this operation not to cause an underestimation of the WCET a value is added to the WCET for each flush (how much that is to be added is not stated clearly). Compared to our work this method does not produce *safe* values, and introduces substantial pessimism.

An approach for deriving an approximation of the WCET for programs executing on hardware with an unknown timing model is presented in [CBG00]. This work requires that performance monitors are available "on chip" to convey information on the number of pipeline stalls, cache misses etc. Based on this and the measured execution time, an approximation of the execution time of basic blocks is derived.

Our work does not require any performance monitors, and hence it can be applied to a wider range of hardware architectures (even though we do not handle caches).

The problem of finding the WCET of programs by dynamic testing methods has been studied by a number of research groups. Approaches using genetic algorithms seem to be the most successful ones for deriving the worst-case input for programs, see e.g. [PN98, PB99]. However, no guarantee can be given regarding the safeness of the measured execution time (even though they usually are quite tight).

Static analysis methods, e.g. [EE99, LBJ+95], use either cycle-level simulators or other models of the hardware to compute the execution time of individual basic blocks. For pipeline analysis the worst-match between basic blocks are found, and hence also the worst timing for the blocks. The obtained values are used in high-level analysis to find the WCET of the entire program. Since, our work is based on the assumption that accurate hardware models are non-existing or difficult to construct these approaches are not sufficient for us. However, if there are such models these approaches are perfectly valid.

2.9 Conclusion

We have introduced the theoretical foundation for a method deriving the execution time of program sub-paths. Combined with high-level analysis and calculation tools, this method can be used to derive the worst-case execution time of programs. The method relies only on the target hardware itself for retrieving timing information, making it applicable to a wide range of processors, even those for which timing models are not available. The approach is in its present form limited to architectures satisfying the sum of basic-block property, with the extension that simple pipelines can be handled. A case-study has been performed that shows that the method does work and is useful, which also included development of prototype tools for the method.

There are several directions in which we would like to pursue this work. First of all, the technique should be validated by applying it to a substantial case-study. This will most likely require further development of our prototype tools and some form of integration with high-level analysis and calculation tools. On the more theoretical side, we would like to extend the method to also handle basic blocks whose execution time is not constant, e.g., caused by division on some architectures. This can probably be handled by using, for example, the least square method when solving the eqation system, and then also compute a confidence value on the derived values. If this work is successful we might integrate the method with the high-level analysis tool presented in [EES+99].

Acknowledgements

We are grateful to the invaluable help provided by Hans Hansson and Henrik Thane with writing the paper on which this is based.

We also wish to thank Joakim Adomat for valuable help with modifying the μSYSA processor, Mohammed El Shobaki for helping with various issues related to the processor, and Thomas Nolte and Dag Nyström for letting us use their VHDL code for the processor (even though we had to spend quite some time fixing bugs in it).

Bibliography

[Alt96] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 102–107. IEEE Computer Society Press, June 1996.

[ARM] ARM (Advanced Risc Machines). Home-page: `http://www.arm.com`.

[ASU86] Alfred V. Aho, Ravi Sehti, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[CBG00] Matteo Corti, Roberto Brega, and Thomas Gross. Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems. In *Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems*. Springer, 2000.

[CI95] David Coward and Darrel Ince. *The Symbolic Execution of Software.* Chapman & Hall, 1995.

[EE99] Jakob Engblom and Andreas Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 1999.

[EES$^+$99] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Towards Industry Strength Worst-Case Execution Time Analysis. Technical Report 99/2, ASTEC, 1999.

[EES$^+$00] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-Case Execution-time analysis for Embedded Real-time Systems. *Submitted for publication*, May 2000.

[EG97] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. 3^{rd} International European Conference on Parallel Processing*, pages 1298–1307. Springer Verlag, August 1997.

[Eng97] Jakob Engblom. Worst-Case Execution Time analyzis for Optimized Code. Master's thesis, Department of Computer Systems, Uppsala University, September 1997. DoCS MSC Thesis 97/94.

[Eng99] Jakob Engblom. Static Properties of Commercial Embedded Real-Time Programs, and Their Implication for Worst-Case Execution Time Analysis. In *Proc. 5^{th} Real-Time Technology and Applications Symposium (RTAS'99)*, pages 46–55. IEEE Computer Society Press, 1999.

[EY97] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 598–604, 1997.

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Department of Computer Engineering, Mälardalen University, May 2000. MRTC 00/10.

[LBJ+95] Sung-Soo Lim, Young Hyan Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), July 1995.

[LHT00] Markus Lindgren, Hans Hansson, and Henrik Thane. Using Measurements to Derive the Worst-Case Execution Time. In *Proc. 7^{th} Real-Time Computing Systems and Applications.* IEEE Computer Society Press, 2000.

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc.* 16^{th} *Real-Time Systems Symposium*, pages 298–307. IEEE Computer Society Press, December 1995.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[OS97] Greger Ottosson and Mikael Sjödin. Worst Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1997.

[PB99] P. Puschner and A. Burns. Time-constrained sorting - a comparison of different algorithms. In *Proc.* 11^{th} *Euromicro Conference on Real-Time Systems*. IEEE Computer Society Press, 1999.

[PB00] P. Puschner and A. Burns. A Review of Worst-Case Execution Time Analysis. *Real-Time Systems*, 18(2/3):115–128, May 2000.

[PF99] Stefan M. Petters and Georg Färber. Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proc.* 6^{th} *International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 1999.

[PK89] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, pages 159–176, September 1989.

[PN98] P. Puschner and R. Nossal. Testing the results of static worst-case execution time analysis. In *Proc.* 19^{th} *Real-Time Systems Symposium*. IEEE Computer Society Press, 1998.

[ZAHM97] Hong Zhu, Patrick A.V, Hall, and John H.R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4), December 1997.

Chapter 3

Deriving Reliability Estimates of Distributed Real-Time Systems by Simulation

Abstract

Industrial deployment of academic real-time techniques still struggles to gain momentum due to the non-familiarity of the industry with schedulability analysis, as well as the lack of appropriate commercial tools. More over, it is imperative that academia realises the extent of pessimism in the proposed techniques, which often makes them less attractive to systems developers.
The possible trade-offs in timing guarantees vs. reliability is one such key area which needs closer study and scrutiny. There is a need for less stringent guarantees in order to avoid costly overdesigns of systems.
In this paper, we present a framework and simulation based methodology for reliability analysis of distributed real-time systems. We have developed a tool which is quite versatile and can accommodate varied task models, network topologies and scheduling paradigms. The tool is illustrated by a comprehensive case-study.
Since our method is based on simulation, which is a standard practice in many industrial projects, we believe it will be more comprehensible and acceptable to the industry.

3.1 Introduction

Despite the advances in schedulability analysis in recent years, the majority of today's real-time systems have not been verified with these methods, nor are schedulability analyses used in the development of most new systems. Yet, these systems are being used to control safety-critical equipment. Deployment of research results in industry is not trivial, even in cases - such as schedulability analysis - when benefits are obvious (at least for academic researchers).

Two of the apparent reasons why the industry is not applying the developed methods are: 1) the lack of commercial tools with professional support, and 2) difficulties in verifying existing implementations; the abstractions used in scheduling analysis do not always match well with how systems are typically implemented, often requiring manual steps that increases the likelihood of introducing errors in the analysis, e.g., how do we map a system implemented using VxWorks to a schedulability model? The services real-time operating systems provide can be quite hard to correctly map to schedulability models.

Another less obvious, but important reason for the non-popularity of developed real-time techniques is their pessimism, which is needed in order to guarantee that requirements are satisfied in worst-case scenarios. However, whether the worst-case scenarios will occur during execution or not is rarely known (or investigated). Previously [HNP00a, HNP00b], we have provided motivations for shifting focus towards the overall system reliability rather than considering only the worst-cases. There are three main motivations. Firstly, there is a large class of "soft" applications which cannot use real-time scheduling analyses, since the provided guarantees are too strong; soft applications do not require strong guarantees. Secondly, the pessimism of hard real-time scheduling analysis often requires systems to use substantially more resources than what is needed under "normal" operation, which for an embedded system may require more advanced and costly processors; making the product more expensive and less competitive. The third motivation is that most systems are resilient to deadline misses to some extent (hard real-time scheduling considers a single deadline miss to be a severe failure). For example, control systems that cannot tolerate a single deadline miss are considered instable [Mar98]. Hence, there is a need for more complex and application specific failure semantics, allowing failures to, for instance, be defined as multiple deadline misses or specific patterns of events.

Not only is it the pessimism of scheduling analysis that require systems to use more resources, but also the pessimism in worst-case execution time (WCET) analysis. One of the problems with WCET analysis is the lack of tools. To our knowledge there are no commercial tools that can derive the WCET for tasks, not even on simple processors. Furthermore, WCET analyses do not consider functional dependencies between communicating tasks, e.g., if tasks A and B exchange data then there may be functional dependencies which makes: $WCET(A) + WCET(B) > WCET(A + B)$[1], yet another source of pessimism.

[1] $WCET(x)$ denote the WCET of x when analyzed in isolation and $WCET(x + y)$ the WCET when x and y interactions are considered.

Hard real-time scheduling often assumes that the WCET of each task is known. Such an assumption works perfectly well when doing basic research, but for industry this renders almost all the nice theories useless. Since there are no tools that can compute the WCET, how can we then use methods which assume them to be known?

Above we have motivated the need for methods applicable for cost conscious industries which manufacture products with real-time requirements, but which cannot afford the pessimism and difficulties in using traditional (hard) scheduling analysis. Furthermore, what industries often are interested in is computing the system reliability, rather than showing that some hard real-time requirements are satisfied. However, it should be borne in mind that our method is not intended to be an alternative to hard real-time scheduling, but rather a complementary one. Thus, in this paper we provide a method, which based on simulation, computes the reliability for individual application functions (implemented by a set of related tasks and messages) in a distributed system. The method is significantly simpler to apply than existing scheduling analysis, making it attractive also to engineers lacking a thorough education in real-time analysis.

In tune with earlier work [HNP00a] we also present a scheduling framework, which is applicable to a wide range of applications that do not necessarily need a 100% guarantee that all deadlines will be met, but rather an estimation of the likelihood of failures, e.g., a reliability of 10^{-9} failures/hours (systems that with a positive probability eventually will fail at some point in time are the only systems that can be built after all). The specific contributions of our simulation based method are that:

- It extends the class of applications that can be analyzed with respect to timing.
- It provides a simpler way of analyzing complex distributed real-time systems with respect to their timing behavior; the complexity in terms of dependencies among tasks and messages that the method is able to handle is significantly larger than what can be handled by current hard real-time scheduling analysis.
- It allows more realistic and explicit failure semantics for tasks and messages.
- It provides tighter coupling between analysis models and implementations.

The reasons we use simulation instead of the real system, are that 1) we wish to make a rapid exploration of many scenarios (simulation on a high-end machine is substantially faster than testing the often very limited real system), 2) we only want to look at timing behavior, hence we can use a model that is more abstract than the real system (which also speeds up the exploration further), and 3) we want it to be possible to perform the analysis at an early stage of system development, i.e., before the implementation is completed.

We believe that this approach is likely to be more attractive to people working in industry compared to hard real-time scheduling analysis, which is not yet industrial practice. Simulation, on the other hand, is extensively used in industry to verify many other system properties, e.g., control performance.

The outline of the paper is as follows. Section 3.2 presents our framework for computing reliability, explains to which kind of systems the method can be applied, and provides information on the prototype simulator used to compute the reliability estimates. A case-study that further motivates the usefulness of our approach is presented in Section 3.3. Related work is reviewed in Section 3.4 and conclusions are provided in Section 3.5.

3.2 Framework and Methodology Overview

In this section we present a framework for computing the reliability of (application) functions executing in distributed systems. Each function consists of a set of related tasks and messages. The reliability estimate is derived using a simulation based approach, rendering it useful to any system that can be simulated (i.e., all realistic systems).

In this paper we consider reliability with respect to timing failures only, i.e., we do not consider the reliability related to functional correctness of software or to hardware sub-systems. These can, however, easily be added since all that is required is to compose reliability estimates; which amounts to multiplication for independent estimates.

3.2.1 System Configuration

The framework does not target a specific task model, scheduling algorithm or bus topology. It can essentially be used for any distributed system. Here we will briefly present the type of target systems our prototype simulator currently is capable of handling. The simulator is implemented in C++, exploiting the benefits of class hierarchies and templates in order to simplify incorporation of new behaviors.

A *system* consists of a set of *nodes*, which are connected by one or more *buses*. On each node there is a set of *tasks* and on each bus a set of specified *messages* can be sent. Each node has a *kernel* that uses a particular *scheduling algorithm*; different scheduling policies can be used on different nodes. The kernel provides system calls for the tasks to use. This is the organization that is used internally in the simulator depicted in Figure 3.1. We refer to nodes, buses and tasks as

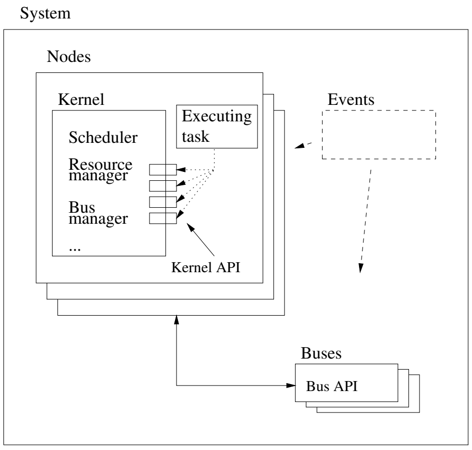

Figure 3.1: The representation of systems in the simulator

entities. The commonality among entities is that they all have behavior which can be simulated.

Probes can be attached to entities to observe their state changes. The probes are mainly used to determine when errors and failures occur. Probes extract data needed to compute the overall reliability of a function.

Tasks and messages are scheduled according to some scheduling policy. The buses can have different bandwidth, e.g., one bus running on high speed and another running on low speed. The clocks on the nodes may be synchronized (using some clock synchronization algorithm) or un-synchronized.

Nodes

Each node has a kernel. Currently, the following kernel functions are available: *take mutex*, *leave mutex*, *relative sleep*, *absolute sleep*, *send message to queue*, and *wait for message.* These functions are also commonly found in commercial real-time operating systems. Hence, we can easily map system calls in say, a system running VxWorks, to a call in our "simulated" kernel.

A node has a queue of tasks that are ready for execution. Upon creation of the node the programmer/user selects how this queue should be sorted, i.e., the scheduling policy is specified.

Clock accuracy is specified when creating a node. This value specifies the speed of the local clock, relative to a perfect clock (by setting this value to 1 the local clock becomes a "perfect" clock).

Tasks

Each task has a behavior which is specified by a sequence of states. Basically there are two kind of states: 1) *call* states, and 2) *execute* states. When in a *call* state, tasks use one of the services provided by the kernel; during this state the task cannot be preempted. In an *execute* state the task can be preempted (determined by the scheduling policy), and the duration of the state is decided by a *distribution object*. This is also a major difference compared to hard real-time scheduling. We use existing task models, but replace worst-case values with values from some distribution. Distribution objects are further described in Section 3.2.3.

Tasks have an active priority, which is set according to the task model in use. Kernels implementing priority inheritance schemes can change the active priority. Usually the kernel sorts the queue of ready tasks based on their active priority. For the case study in Section 3.3 we have used the fixed priority scheduling with support for sporadic tasks.

Probes

The probes extract data that enable the computation of reliability estimates of functions in the system. In order to compute the reliability of a function the probe must extract the following data: number of failures, and number of instances of the function that has been executed. For example, if the function for which the reliability is to be computed is a task, then we could count the number of deadline misses and number of task instances executed, and from that compute a reliability estimate. Of course, probes can also be used to extract other data from the system, e.g., response times and start jitter.

Bus interference

The bus(es) which connect the nodes in the system can be subject to different patterns of interference. This can cause messages to be corrupted, which require them to be re-transmitted. We consider such interference, but also the time that is required for the bus to recover from such an error (in much of the same way as in our earlier works [HNP00a, HNP00b]).

When creating a bus, its *bandwidth*, *recovery time*, and connected nodes are specified. The recovery time can for a Controller Area Network (CAN) be the time required for error signaling. From this we can during simulation compute message transmission times and recovery time from transmission failures. The bus also has a scheduling policy enabling the simulation of any popular networks, e.g., TDMA, CAN, and token ring networks.

The simulator injects interference to the bus by calling the `Zap(duration)` method, which causes transmission failures on the bus for the specified duration. Any interference pattern can be created in this way.

3.2.2 Reliability Estimation

The core of the approach is to simulate the system for a period of time and observe timing failures. From the number of failures and number of executed instances of a function we can compute its overall reliability.

The state space of a simulation is usually too big for exhaustive simulation to be effective [HNP00a]. Instead we select a finite number of samples, and from these we make statistical predictions about all instances. By simulating the system n times for a finite time period t, and extracting data on number of failures and number of executed instances of the function, we get a set of failure probabilities $f_1, f_2, ..., f_n$, where $f_i = \frac{number\_of\_failures}{number\_of\_instances}$ for simulation run i. From that we compute the reliability $r_i = 1 - f_i$.

By applying the central-limit theorem [Gun89] we can compute the overall reliability R for time period t, with say a 99% confidence interval, I.

$$Q = \sum (r_i - \overline{r})^2,\ s = \sqrt{\frac{Q}{n-1}},\ d = \frac{s}{\sqrt{n}}$$

$$I = (\overline{r} - 2.66 \cdot d,\ \overline{r} + 2.66 \cdot d)\ ,\ R = \overline{r}$$

All that remains is to decide the time period t of the simulations runs (under the assumptions of the central-limit theorem), which is detailed below.

Compared to hard real-time scheduling we do not have a finite period of time, such as the least common multiple (LCM) of the tasks periods, after which the system repeats its behavior. Since, in our model tasks need not be strictly periodic, task behaviors are determined by how and when they make system calls. In general it is not possible to find a time after which the systems repeats itself. The problem resembles software testing, where there also is a problem of when to stop. For software testing, test coverage criteria are decided on [Tha00], e.g., execute all statements once or execute all branches in the program (similar criteria also exist for concurrent activities, such as task execution). Testing ends when, say, 99% of these have been tested successfully. End criteria can also be used for simulations, for example, end simulation when 2% of the state space has been explored.

In the worst-case, simulation rounds must last the entire mission time of the system. Mission times depend on the application, but as an example the mission time of a typical automotive system is 8 hours, and may for a space probe be up to 15 years or more. Simulating a system for 15 years repeatedly is not practical. (Note that a mission time of 15 years does typically not require 15 years of simulation, but it can still be substantial). There are however cases in which the simulation time can be reduced. If it is possible to find a point in time after which the system repeats itself, then it is sufficient to simulate this period of time, e.g., for a strictly periodic system it is sufficient to simulate the period of time that equals the LCM of the tasks.

The assumption on the function that is simulated is that all its tasks and messages must be related. For example, we can compute the reliablity of an automotive ABS-braking system function using this approach. If it is desired to compute the overall system reliability, then the reliability of all functions in the system must be computed individually and then combined into an overall value. Such a composition will typically involve reducing the system into meaningful series or parallel blocks and using standard equations for composing reliability block diagrams [Pra96].

It is worth noting that if the tasks' timing attributes are set to worst-case values and the system is exhaustively simulated, then the result obtained coincides with the results of hard real-time scheduling analysis (given that such analysis can be performed). The simulation based approach is, however, likely to be slower for those cases, since its model of the system contains more details. Thus, our approach is more general than hard real-time scheduling, but maybe not equally efficient in providing a priori guarantees. On the other hand, the simulation based approach can also convey information which analysis cannot, e.g., number of deadline misses for a task within the LCM. Simulations can also guide engineers in finding bottlenecks in the system. This is not as straightforward using hard real-time analysis (which essentially provides "yes"/"no" responses).

3.2.3 Execution Time Distribution

In a previous paper [HNP00a] we assumed that tasks' execution times were uniformly distributed. This is for most applications not the case. Here we present an extension which allows more realistic distributions for task execution times.

By using execution times from distributions rather than worst case values, we can simulate all possible execution scenarios that can occur during run-time. Specifically, race conditions, which may or may not be found using worst case values, can be found using our approach. Note that race conditions may have a major impact on the timing behaviour of an application.

The execution time distribution of software depends not only on the code itself, but also on the environment in which the software is being used. Therefore, the same distribution may not be appropriate for every use of a function/task. How to obtain these distributions is one of the problems we are faced with.

Execution time distributions can be obtained by logging input data when the task is running in the real system (and its designated environment), and measuring the execution time for the input data distribution. The more time spent on deriving execution time distributions that reflects the timing behavior of the software, the more we can rely on the computed reliability estimates. Fortunately, this does not mean a lot of extra tedious work, since much of this can be carried out in parallel with module testing of the real system.

Input distributions for new applications are normally not known. Hence, we have to resort to other methods in those cases. Possible approaches are: using knowledge from domain experts to derive estimations of the distribution, or play around with different scenarios that we may expect, which will give us a feel of the systems timing behavior.

A problem with measuring execution time is that there is rarely time to explore all input data combinations. There can, in effect, be tasks whose execution time is shorter or longer (or both) than the measured. If we want to model the real distribution correctly we have to interpolate between samples, and perhaps also extrapolate beyond the measured values. However, finding the "correct" way to extrapolate and interpolate execution time is hard. In [BE00] an approach to extrapolation using statistical methods is presented.

We will not consider interpolation or extrapolation between sampled values further, since it is outside the scope of this paper. In our case study we will use values from a discrete distribution, where each sampled value in the distribution represents one execution path. Such distributions are applicable on hardware platforms that do not use pipelines or caches.

Functional dependencies between tasks, which may affect the execution time of the tasks, is not considered in this paper. Since, we as of today do not know how to model these dependencies, nor do we know to which extent there are such dependencies. Not considering dependencies may introduce some pessimism.

3.2.4 Failure Semantics

A single deadline miss is considered to be a severe failure in most hard real-time scheduling analysis, although there are some exceptions [BB97, KS95]. This is rarely the case in control applications [Mar98]. In fact, a controller which cannot handle a single deadline miss is typically considered to be instable. Therefore, we allow more relaxed failure semantics, which are defined on a per task basis.

The failure semantics that we consider in this paper is a combination of the allowed number of consecutive deadline misses and the allowed number of deadline misses in a specified interval. For example, a failure for a specific task can be two consecutive deadline misses or three deadline misses within 15 task instances.

In the simulator we currently have a probe which extracts the above data. Upon creation of the probe, the number of allowed consecutive deadline misses, number of allowed deadline misses in an interval, and the interval length is specified. The data collected by this probe is used in the reliability computation. Other probes, with different failure semantics, can also be incorporated into our framework.

3.2.5 Simulator

The nodes in our simulation system need not be perfectly synchronized. Hence, events can occur at any time during simulation. Thus, it is not sufficient to use time driven simulation in which a clock is advanced by one tick each simulation step. Instead, we use an event driven simulation engine, as described in the following.

Each entity that can be simulated (bus, node, task) must be able to perform the following two methods: *NextEventTime* and *SimulateNextEvent*. The *NextEvent-Time* method returns the time for the next event the entity will perform (given its current state), and *SimulateNextEvent* performs the next event on the entity, which also can change state on other objects in the system (causing them to report a new time as their *NextEventTime*).

All that the simulator needs to do is to ask the entities for their next event time, and then execute the one with lowest value. Just before calling the entity, the system global time is advanced to the time of the event.

3.3 Case-Study

To illustrate the concepts and benefits of the framework we apply it to a simple case-study. The case-study is based on an assignment in a real-time systems course being held at Uppsala University [Erm00]. This is really just a simple "toy" example, but it contains the main ingredients of a real system, even though a real system is significantly more complex, both in terms of size and interactions among tasks.

3.3.1 System Characteristics

The assignment is to implement a system that sorts glass and metal marbles rolling down a slide, as illustrated in Figure 3.2. The system has two sensors and two actuators. The sensors are located along the slide; a magnetic sensor detects

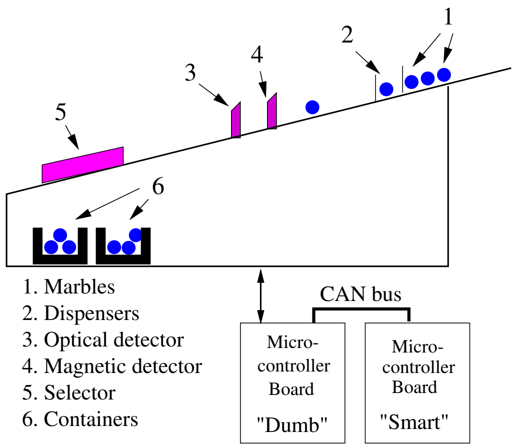

Figure 3.2: Sketch of the marble sorter system

metal marbles and an optical sensor detects any marble passing by. The dispenser actuator controls when a marble is released. The selector controls into which tray the marble is being put (the possibly "hazardous" state is when a glass marble is put into the tray for metal marbles). The real-time problem is to guarantee that the selector is put into the right position before a marble reaches it (and that it remains in that position until the marble has passed).

The computer system consists of two nodes connected by a CAN-bus, see Figure 3.2. The node (*dumb*), connected to the marble sorter, is responsible for relaying sensor values to the other node (*smart*) and to control the actuators. No control decisions are taken at the dumb node. The preemptive fixed priority scheduled TPK operating system (by NRTT, UK) is used on the nodes.

A typical student solution to this exercise contains one periodic task (that releases marbles), and all other tasks are sporadic, with inter-arrival times mainly determined by the periodic task. A solution to this assignment is illustrated in Figure 3.3. The solution consists of five tasks, two located on the smart node, and three on the dumb node. The bottom two tasks are used for loading and releasing marbles, the other three retrieve information about the marbles and decide the position of the selector.

The timing in this system is such that:

- Marbles pass the magnetic sensor after 0.36–0.38s and the optical sensor after 0.38–0.43s.
- The selector must have been switched at 0.52–0.55s after the release of a marble.

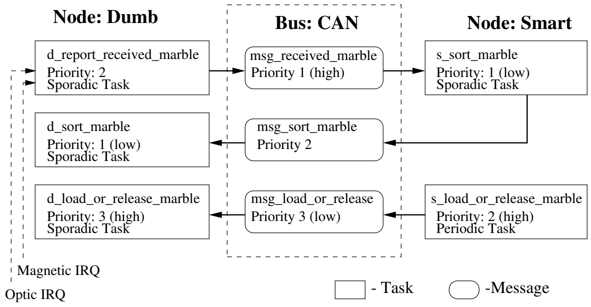

Figure 3.3: Tasks, messages, and attributes for a solution to marble sorter

The timing requirements have been obtained by extensive testing of the system. We use 0.52s as the critical time (end-to-end deadline) in this system.

3.3.2 Reliability Estimation

In deriving a simulation model of the system in Section 3.3.1 we look at the implementation and create a sequence of states for each task. In each of these states the tasks either do a system call or execute code for a while (according to some distribution).

The pseudo-code for the task *s_load_or_release* is:

```
while(1) {
  load marble     // send event to dumb
  delay_from_release(MARBLE_LOAD_TIME)
  release marble  // send event to dumb
  delay_from_release(MARBLE_RELEASE_TIME)
}
```

Our simulation model of this periodic task becomes:

```
state 1: execute[1.018ms]
state 2: MsgSend("toDumbLoad")
state 3: releaseInPeriod(MARBLE_LOAD_TIME)
state 4: execute[1.357ms]
state 5: MsgSend("toDumbLoad")
state 6: waitForNextPeriod()
```

First the task executes for 1.018ms (there is only one path in this task, so execution time distributions are not needed). Secondly, the task controlling the release of marbles is notified by sending a message to it. In the third state it waits until period start + `MARBLE_LOAD_TIME`. Then it executes for 1.357ms. Finally, it sends another message to the dumb node (we do not model the task's functional behavior, i.e., whether it is a glass or metal marble doesn't matter in the simulation).

There is a tight coupling between the implementation of the task and our simulation model. It can be made even tighter by writing call states with the same name and arguments as the counterparts in TPK, and then using these call states to describe the behavior of the task. If we introduce this software layer, some simulation models may even be automatically generated, which is also one of our future aims for the simulator.

The pseudo-code for the task *d_load_or_release* is:

```
while(1) {
  wait for message on queue "toDumbLoad"
  read message
  if load message => load marble
  if release message => release marble
}
```

The simulation model becomes:

```
state 1: MsgReceive("toDumbLoad")
state 2: execute[1.238ms]
state 3: MsgReceive("toDumbLoad")
state 4: execute[0.883ms]
state 5: MsgSend("toReport")
```

This task is sporadic, and in its first state it waits for a message to trigger its invocation. In the second state a marble is loaded into the dispenser. Since we know that *load* and *release* messages are alternating we can we can unrol the loop. Therefore the task waits in its third state for another message to trigger it (corresponding to the second iteration of the while statement). It executes some more before it releases a marble (MsgSend). The code for the other tasks in the system are similar, except for some execute statements which uses values according to a distribution. Details for each task can be found in [LHNP00].

To model the time between the release of a marble and interrupt invocation (see Figure 3.3), we use a *delay bus*. The purpose of a delay bus is to delay the delivery of a message according to some distribution, in this case a delay of 280–330ms (actually this "bus" models part of the physical environment).

Now, all tasks and messages in this system together perform the function of sorting marbles. Since, all tasks are related we can derive a reliability of the system using the framework presented in Section 3.2.

The mission time for this system is possibly infinite, but since the trays can only hold a limited number of marbles it can be reduced. After a *batch* of marbles has been sorted the system reaches a state after which it will start all over again. The trays need to be emptied and the dispenser needs to be refilled. The simulation time for this system can therefore be reduced to a batch of marbles, consisting of, say, 100 marbles.

An end-to-end deadline probe is inserted into the system to monitor the time from the release of a marble, to the time of switching the selector. The probe extracts data on number of failures and executed instances (the maximum allowed end-to-end time is 0.52s).

By simulating this system we obtain the following results (100 marbles in each sample):

$n = 200$ (samples)	confidence 99%
reliability 0.9357	conf. interval $(0.9308, 0.9406)$.

The result is not good at all (about 7 of 100 batches will be incorrectly sorted).

Now let us assume that instead of sorting marbles, our application deals with batches of colored and transparent glass bottles that should be sorted for recycling. For this application it might not be as important that all bottles are sorted correctly, but rather that a certain percentage must be sorted correctly, i.e., we have a different failure semantics. Remember that for control-applications it is important that not too many deadline misses occur in sequence and not too many within an interval of task invocations.

To see how the failure semantics affects the reliability we run the same simulation as above, but change to a different failure semantics. By allowing three end-to-end timing violations in sequence and seven within an interval of 30 bottles (just as an example), we obtain the following result:

$n = 2000$ (samples)	confidence 99%
reliability 0.9984	conf. interval $(0.9976, 0.9992)$

Now approximately 2 out of 1000 batches will have to be discarded. Finding the correct failure semantics for the application being analysed is crucial for reliability estimations (and any other form of analysis).

By comparison, if we run the same simulation as above with worst-case values instead of values from distributions we get the following result:

$n = 2000$ (samples)	confidence 99%
reliability 0.03000	conf. interval $(0.029988, 0.03001)$

According to this, the system is in really bad shape even though the "forgiving" failure semantics was used (this result says that on an average only 3 of 100 batches will be correct). By looking more closely at the simulation runs, we see that the first two instances are not considered to be failures (as per our failure semantics). Hence, increasing the number of bottles per batch will reduce reliability.

3.4 Related Work

Simulation for evaluating real-time systems behavior has been reported elsewhere. Here we review some of the most relevant results.

There are many commercial tools capable of simulating the functional behavior of systems, e.g., VxSim for VxWorks [Win99] and virtual target for VRTX [Mic96]. However, there are only a few simulators that considers timing.

STRESS is a simulator for hard real-time systems [ABRW94, Mik92]. Its primary use is for evaluating real-time systems, and to aid engineers in pinpointing timing errors. It has a language for describing the behavior of tasks (with, for example, variables and loops), and uniformly distributed execution time distributions. Access is provided to semaphores and mailboxes, and multi-processor nodes are supported. STRESS is more sophisticated compared to what our simulator currently is, but no guarantees or the like are automatically derived from the simulation.

DRTSS is a simulation framework for hard and soft real-time systems [SL96], quite similar to STRESS. One difference is that it is capable of searching for minimum values of certain parameters, such as deadlines, periods, release-times.

The simulator tool described in [Cer] focuses on how real-time kernels and their scheduling policies affect control performance. Of specific interest to our work is the fact that a single deadline miss is not always catastrophic, i.e., different failure semantics for different kinds of application is needed.

3.5 Conclusion

Verifying distributed real-time systems' temporal properties is hard. Especially if the implementation uses message queues and other system calls in RTOSs, which is common in many industrial applications. To enable the estimation of these systems' reliability we have proposed a simulation based framework which can handle in essence any distributed system. We believe that our approach is easier to use as compared to the traditional hard real-time scheduling techniques.

The core idea of our approach is to simulate the system for a finite time period, and then calculate the failure rate. By taking a number of random samples from the simulations we can compute the overall reliability of a function in the system. The results that can be obtained are more general compared to results from hard real-time scheduling analysis.

The advantages are that the analysis is simple, and that it is easy to map an implementation in essentially any RTOS to a faithful simulation model. This step is not always easy in hard real-time scheduling. We have shown the usefulness of our approach using a small case-study and a prototype simulator that we have implemented.

Future work includes extensions of the simulator and further validation of the method by applying it in the development of larger scale industrial systems.

Bibliography

[ABRW94] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. STRESS: A Simulator For Hard Real-Time Systems. *Software - Practice and Experience*, 24(6):543–564, 1994.

[BB97] Guillem Bernat and Alan Burns. Combining $\binom{n}{m}$-Hard deadlines and Dual Priority Scheduling. In *Proc. 18^{th} Real-Time Systems Symposium*. IEEE Computer Society Press, 1997.

[BE00] Alan Burns and Stewart Edgar. Predicting Computation Time for Advanced Processor Architectures. In *Proc. 12^{th} Euromicro Conference on Real-Time Systems*. IEEE Computer Society Press, 2000.

[Cer] Anton Cervin. Towards the Integration of Control and Real-Time Scheduling Design. Licentiate Thesis. Department of Automatic Control, Lund Institute of Technology, Sweden, 2000.

[Erm00] Andreas Ermedahl. The Marble Sorter Assignment, June 2000. `http://www.docs.uu.se/~ebbe/realtime/marble_sorter/`.

[Gun89] Gunnar Blom. *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 1989. (Note: a Swedish book on statistics).

[HNP00a] Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Hard Real-Time in a Soft World. Technical report, MRTC, May 2000.

[HNP00b] Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Reliability Modelling of Time-Critical Distributed Systems. In *Formal Techniques for Real-Time and Fault-Tolerant Systems (FTRTFT)*. LNCS, Springer Verlag, September 2000.

[KS95] Gilad Koren and Dennis Shasha. Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips. In *Proc. 16th Real-Time Systems Symposium*. IEEE Computer Society Press, 1995.

[LHNP00] Markus Lindgren, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Deriving Reliability Estimates of Distributed Real-Time Systems by Simulation. Technical report, MRTC, Sweden, September 2000.

[Mar98] Martin Törngren. Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. *Real-Time Systems*, 14:219–250, 1998.

[Mic96] Microtec. *Spectra Backplane Concepts*. 1996. MGC PartNo. 200215 Microtec PartNo. 102008-001.

[Mik92] Mike Richardson. The STRESS Hard Real-Time Simulator. Technical report, Real-Time Systems Research Group, York University, 1992.

[Pra96] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996. ISBN 0-13-057887-8.

[SL96] Matthew F. Storch and Jane W.-S. Liu. DRTSS: A Simulation Framework for Complex Real-Time Systems. In *Proc. Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, 1996.

[Tha00] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, Sweden, 2000.

[Win99] WindRiver. Tornado User's Guide (UNIX version), 1999. `http://www.wrs.com/products/html/manuals.html`.

Chapter 4

Case-study: Using Simulation to Verify Real-Time Properties

4.1 Introduction

The purpose of the case-study reported here is to identify the strengths and weaknesses of the method proposed in [LHNP00] (Chapter 3 in this thesis), and to find future directions for this research. We do this by applying the method to a small case-study of a sub-system of a real industrial system. We are not allowed to reveal the company's name, nor any details about their system. Therefore, we have changed the names of the components in the system, but the structure of the system is kept intact. We assume that the reader is well familiar with our simulation method prior to reading this report.

From the implementation of the sub-system we derive an abstract model of the system; which we then simulate to determine if certain properties hold. More specifically in this case-study we are interested in determining the maximum and minimum length of a specific message queue. The length of this queue is entirely dependent on the real-time behavior of the tasks in the system. To be able to perform simulations we also have to measure the execution times of the tasks that define the sub-system.

The outline of this report is as follows: Section 4.2 presents the system behavior and the simulation model of the system. In Section 4.2.2 we present the simulation results. Section 4.4 identifies future directions for research and strengths of the method. Finally, in Section 4.5 we make some concluding remarks.

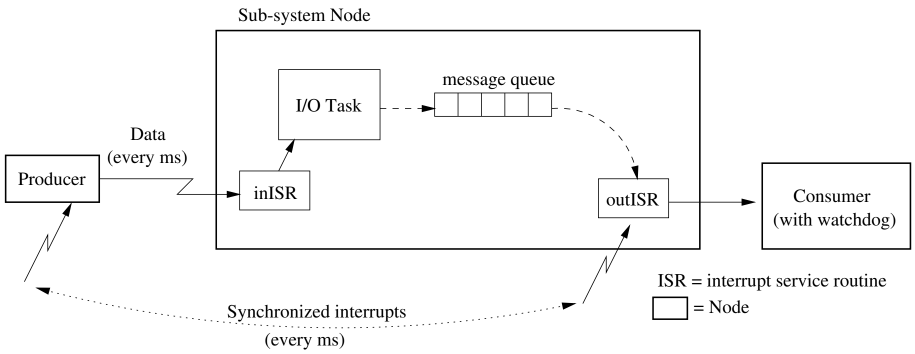

Figure 4.1: The components in the system

4.2 System Behavior

Here we present the different components of the sub-system we are analyzing. There are three nodes in the system; one that produces data (producer), one that consumes data (consumer) and then the sub-system node that we analyze in this case-study. The sub-system takes data from the producer and sends it to the consumer, as illustrated in Figure 4.1. The producer and our sub-system node has a global time base that controls when timer interrupts occur at the nodes. The timer ISR (interrupt service routine) on the producer node generates new data that is sent to the sub-system node. This causes the *inISR* to be invoked on the sub-system node, which basically identifies that new data is available for processing and informs the *I/O Task* on the new data. The *I/O Task* takes the new data and puts it in the message queue (FIFO).

At about the same time as the producer receives the timer interrupt the sub-system node also receives it. This causes the *outISR* to be invoked, to take the first data in the queue and send it to the consumer. At an abstract level this is the functional behavior of the sub-system node. In addition to the above tasks and ISRs there are two periodic tasks (A and B) with higher priority that can preempt the I/O task, and may cause the I/O task to be delayed. We will return to their behavior later.

If the consumer does not receive data every ms a watchdog is released that performs a reset on the system. This reset is not desired in any way, and can even result in considerable damages. The problem is now to make sure that data is sent to the consumer every ms.

Besides the problem of making sure that data is sent every ms, we must also make sure that the data being sent is not too old. Aged data causes the system to lose performance. The age of the data is determined by the length of the message queue (the delay equals the length of the queue in milliseconds). For high performance we must make sure that the message queue contains as few elements as possible, and that a read never occurs on an empty queue.

To solve this problem the consumer is disabled at system startup until the message queue contains four data elements. The question is now if this is sufficient to avoid resets. We will use our simulation methodology to try to determine this.

It should also be mentioned that the company is not very happy with the current design, but due to various historical reasons it has been implemented anyway. Since, it is a solution that is being used in some of the company's products today, and it is important to make sure that it behaves correctly.

The sub-system node uses an Intel Pentium 3 processor and is running the VxWorks operating system. Besides the above sub-system, the nodes contain application specific code that perform the actual function of the system. This will however not be discussed in this report. We will now briefly describe how the tasks/ISRs are implemented.

4.2.1 System Implementation

The following is the pseudo-code for the implementation of the tasks/ISRs in the system:

```
inISR:    wait for data interrupt
          do some processing
          semaphoreGive(ioSemaphore)

outISR:   wait for timer interrupt
          get data from message queue
          send data to output device

Task A:   do forever
            do some processing
            sleep(100ms)
          end forever

Task B:   do forever
            do some processing
            sleep(500ms)
          end forever

I/O Task: do forever
```

```
    semaphoreTake(ioSemaphore, WAIT_FOREVER)
    get data from input device
    put data in message queue
  end forever
```

The different tasks/ISRs are listed in descending priority order, where inISR and outISR have highest priority in the system and cannot be preempted. The I/O task has the lowest priority. Fixed priority preemptive scheduling is used for the tasks. Also, note that a counting semaphore is used between inISR and the I/O task, and that the function `sleep(x)` called by tasks A and B suspends the task for x ms relative to the time it is called.

4.2.2 Simulation Model

Here we present the simulation model that we have developed for the sub-system. We use the same simulator as in [LHNP00]. First we present the behavioral model of the tasks, and then discuss how we have obtained execution time distributions from the system.

Task Behaviors

The behavior of the tasks has been captured from discussion with our industry contact as well examination of the code for the system and actual executions. The simulation model for the tasks in Section 4.2 become:

```
inISR:    StateSemTake("input")
          StateExecute(distribution("inISR-execution-time"))
          StateSemGive("ioSemaphore")

outISR:   StateSemTake("timer")
          StateSemTake("messageQueue")
          StateExecute(distribution("ourISR-execution-time"))

Task A:   StateExecute(distribution("taskA-execution-time"))
          StateRelSleep(100ms)

Task B:   StateExecute(distribution("taskB-execution-time"))
          StateRelSleep(500ms)

I/O Task: StateSemTake("ioSemaphore")
          StateExecute(distribution("io-task-execution-time"))
          StateSemGive("messageQueue")
```

Since the data being sent via the message queue is not relevant for the properties we are investigating — only the length of the queue is — we have used a counting semaphore for the queue. The initial value of the message queue semaphore is set to four, and all the others are set to zero. In the state *execute* the tasks/ISRs executes according to the distributions we have observed on the target.

The above tasks are put on a fixed priority preemptive scheduling node in the simulator, which has support for the used system calls.

To model the generation of interrupts to the sub-system node we use a separate node, also using fixed priority scheduling. This node has two tasks running with a period of $1ms$. They have the following states:

```
StateSemGive("timer" or "input")
StateRelSleep(1ms)
```

Execution Time Measurements

To derive the execution time distributions for the tasks we have instrumented the system's code and taken between 200 and 1000 samples for each task on the target system. These values are kept in files, which are referred to by the `distribution` function in the presented simulation model.

The execution time has been measured by inserting a call to a function which reads the current time at the beginning of each task, and at the end reads the current time again and compute the difference which is recorded as a sample. Note that this time includes preemption by higher priority tasks and interrupts, since we measure the execution time when the complete system is run. Hence, this is not really a proper way of obtaining these values. However, to simplify the measurements and due to the system configuration and the available slack in the system we decided to use this approach anyway, since small errors in the measured execution time will not cause new execution scenarios for this system.

We did not need to use input data distributions, since the control-flow of the tasks in this system are not dependent on the input data. Input data distributions must be used if the execution time is dependent on the input. Even though the control-flow is independent of the input in this system, the execution time still varied quite much. This variation was observed even for the ISRs which cannot be preempted. The variation can be caused by refresh of DRAM and contention on the bus to which all nodes are connected, and also by different states of the caches and branch prediction units. However, we are not certain on this. These variations would probably not have been identfied by static analysis of the code.

4.3 Simulation Results

We have simulated the presented simulation model in our simulator and observed that the size of the message queue varies between *two* and *four* elements. This coincides with the observations made by the company when executing the target system. However, the simulation model has omitted the following details of the target system:

- there are high priority tasks managed by the operating system, which we have not been able to instrument and model. These tasks are not invoked very often, but can still change the timing behavior of the system.
- the operating system's timer ISR has not been modeled for the, since it too could not be instrumented and modeled.

These differences are somewhat compensated for by including the time for preemption when measuring the execution time. However, this does not model the above behavior accurately, and hence, the simulations may be optimistic due to this.

Since, we have not observed a single failure, i.e., empty queue, during the simulations there is not much reason to compute a reliability estimate for the system (since it is 1).

4.3.1 On why current techniques are insufficient

Over the years, researchers have developed many different real-time schedulability techniques that can be successfully applied to a range of systems. However, these techniques cannot be used on the majority of real-time applications. The main arguments for this are that:

- The way systems are implemented does often not match with schedulability models. For example, in our case-study counting semaphores and message queues are used. Such synchronization methods are not supported by current schedulability models, but are often used when implementing systems.
- All tasks aren't strictly periodic, hence, there is rarely a finite period after which the execution scenarios repeat. Sporadic tasks can be used to model this, but this cause more pessimistic results.
- Schedulability models assume that the worst-case execution time of all tasks are known, but as of yet there are no commercial tools that can derive this. Further, if we look at advanced super-scalar processors such analysis become very complex, if even possible at all, and the result is often very pessimistic.

- Schedulability models only look at worst-case behavior, but does not investigate if the worst-cases occur during run-time or not.

These are some of the motivations for using simulation.

4.4 Evaluation/Future Directions

This section identifies future directions in which the work on deriving reliability estimates for distributed systems should be pursued.

4.4.1 Behavioral Modeling

The case-study in this report is indeed simple, as indicated by the fact that all tasks can be modeled by sequential state machines. It is likely that sequential state machines are not sufficient for modeling more complex systems. Therefore, several more complex systems should be analyzed to find the appropriate expressiveness for the state machines in the simulator.

One extreme would be to use the code as it is and then interface it to the simulator, i.e., make use of the *live task model* [SL96]. We believe that the live task model is not the right approach, rather should a goal model capture only those details that are interesting for the current item under investigation, i.e., more abstract models are needed.

In the live task model it would also be more complex to find execution time distributions, since these need to be found between each function call that changes the state of a task (executing, ready, blocked and waiting).

We believe that it is sufficient to have a state machine model that is somewhere between what is being used now and the live task model. The simulator in [ABRW94] might even have the right abstractions now; it has a primitive language for expressing the behavior of tasks. This should be further investigated. The long term goal for this part of the work should be to automatically generate a simulation model from the code of a task.

It should be noted that for the case-study in this report, less than 10 hours was required to instrument the system and obtain execution time distributions and the behavioral model, even though we did not have any prior experience with the target system. At the same time we were able to capture more of the functional behavior of the system compared to "traditional" methods for schdulability analysis. The close relation to the implementation is a definitive strength of the method.

4.4.2 Execution Time Distributions

Execution time distributions are obtained by exposing the system to the input data it will find in its designated environment, and recording the execution time for those data. The observed samples do, however, usually only represent a small fraction of all the possible execution times. Therefore methods for interpolation between samples and extrapolation beyond the sampled values will be required. Considering also the complexity of current processor architectures it is imperative that such methods are developed.

We don't believe that current methods for worst-case execution time analysis is a feasible approach, since worst-cases will usually require unnecessary over-design of the system. In some cases this over design is motivated, but in the majority of real-time applications it is not. Caches, branch prediction units, and super-scalar processing also make these kind of analysis extremely complex. We believe that measurements on the target combined with statistics is the way to go for these applications. A statistical method for execution time estimation has been presented in [BE00].

Another reason for using interpolation and extrapolation is that if we use only the execution times found during measurements, then we will probably not discover any new execution scenarios during simulation besides the ones seen during the measurements. Since it is most probable that such scenarios do exist, it is also necessary to develop methods for interpolation and extrapolation.

The success of our method is highly dependent on such a technique, in much of the same way as schedulability analysis is dependent on worst-case execution time analysis, therefore this issue should be given high priority in future work.

When performing measurements on the target it is also important that measurements are not performed on isolated tasks, i.e., measurements should be performed when all other tasks in the system also are executing, since this give a chance to capture cache dependencies and the like. Furthermore, it is necessary that the measured execution time for a task does not include preemption by higher priority tasks (as we did in our simulation model) nor blocking by lower priority tasks, but it should include the time for performing system calls. This way the execution time for individual tasks can be obtained.

From our case-study we know that it is easy to find execution time distributions, but that it is harder to find a good way to interpolate/extrapolate.

4.4.3 Simulation Time

Deciding for how long the simulations should last is yet another intricacy of the method. In the case-study we simulated the system for 50 seconds, a simulation which is performed within seconds in the simulator. Hence, a weeks worth of execution time could easily be simulated, say, over a night on a standard PC, which would capture essentially all different phasings of the tasks.

The important thing to note is that tasks are seldom strictly periodic — which often schedulabilty models assume — and hence it is very rare that one can find a point in time after which the execution scenarios repeat.

We believe that it is necessary to use methods similar to statistical testing [How97] to decide for how long time simulations should be performed. There are probably already results in software testing that can be applied for the simulations. This will require future investigations as well.

4.4.4 Scaling

Simulations requires time to be performed! When larger and more complex systems are being simulated it is likely that the simulation time will require too much time, even though we have a sound method for driving the simulations forward. The simulations we have completed so far has required up to one minute of simulation time, as was the case for the marble sorter case study in [LHNP00], the simulations in this paper are completed within seconds.

Since the simulations do not require the target hardware to be used, only a standard desktop computer is required. Possibly could all available PCs in a company be used to perform the simulations. However, care has to be taken so that the execution time distributions remain the same over all the PCs. This issue is resolved by running many simulations on each node.

This issue should currently not be given high priority, considering the current status of the method.

4.4.5 Not Only Reliability

When we initially developed this method the main purpose was to compute a reliability estimate for the system being analyzed. However, the simulations can be used to investigate many other important aspects. For example:

- at an early stage of product development simulations can be used to evaluate different designs. These simulation models can be very abstract.

- simulations are an important aid to investigate the effects of adding new functionality or changing the environment in which the system operates.
- at late stages of development simulations can be used to track down hazardous execution scenarios, and also to derive reliability estimations of the system.

To derive reliability estimations is important, but the above issues are equally important, therefore we should have a wider aim with our methodology (not only reliability).

4.5 Conclusion

This report has identified future directions for our research on developing methods for reliability estimations of distributed real-time systems. We have identified these issues by applying the method to an industrial system.

The conclusion is that the most important issue is to find methods for interpolation and extrapolation of the obtained execution time measurements. The development of such methods is crucial for the success of our simulation method. Secondly, further case-studies should be performed to determine whether the behavioral model of the simulator needs to be extended. This would potentially allow more complex systems to be modeled. Thirdly, software testing methods are to be studied to find a suitable way of determining the simulation time.

From our initial case-studies we also conclude that the method is promising, and that the development should continue.

Bibliography

[ABRW94] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. STRESS: A Simulator For Hard Real-Time Systems. *Software - Practice and Experience*, 24(6):543–564, 1994.

[BE00] Alan Burns and Stewart Edgar. Predicting Computation Time for Advanced Processor Architectures. In *Proc. 12th Euromicro Conference on Real-Time Systems*. IEEE Computer Society Press, 2000.

[How97] William E. Howden. Systems Testing and Statistical Test Data Coverage. In *Proc. 21st International Computer Software and Applications Conference*. IEEE Computer Society Press, 1997.

[LHNP00] Markus Lindgren, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Deriving Reliability Estimates of Distributed Real-Time Systems. In *Proc. 7th Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 2000.

[SL96] Matthew F. Storch and Jane W.-S. Liu. DRTSS: A Simulation Framework for Complex Real-Time Systems. In *Proc. Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, 1996.

Licentiate theses from the Department of Information Technology

2000-001 Katarina Boman: *Low-Angle Estimation: Models, Methods and Bounds*

2000-002 Susanne Remle: *Modeling and Parameter Estimation of the Diffusion Equation*

2000-003 Fredrik Larsson: *Efficient Implementation of Model-Checkers for Networks of Timed Automata*

2000-004 Anders Wall: *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*

2000-005 Fredrik Edelvik: *Finite Volume Solvers for the Maxwell Equations in Time Domain*

2000-006 Gustaf Naeser: *A Flexible Framework for Detection of Feature Interactions in Telecommunication Systems*

2000-007 Magnus Larsson: *Applying Configuration Management Techniques to Component-Based Systems*

2000-008 Marcus Nilsson: *Regular Model Checking*

2000-009 Jan Nyström: *A formalisation of the ITU-T Intelligent Network standard*

2000-010 Markus Lindgren: *Measurement and Simulation Based Techniques for Real-Time Systems Analysis*

UPPSALA
UNIVERSITY

University Printers, Uppsala University, Sweden