FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Achieving an Enhanced Worst-Case Timing Prediction and Performance for Hard Real-Time Code

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Technische Informatik

eingereicht von

Nikolaus Manojlovic
Matrikelnummer 0325354

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Univ.-Ass. Benedikt Huber

Wien, 22.03.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Nikolaus Manojlovic
Obere Augartenstrasse 18/7/3, A-1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschliesslich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22.03.2010 _____

Abstract

Hard real-time computing systems play a crucial role in our society since a considerable number of complex systems rely on processor control that must satisfy specific safety conditions. Meanwhile, hard real-time computing has established itself extensively in the area of safety critical systems, including applications such as nuclear power plants, air traffic control, automotive electronics, robotics, military systems and others. The program code of such hard real-time systems has to meet specific demands with respect to worst-case performance, its so-called execution time (WCET) and, moreover, requires specific properties in order to enable an efficient Worst-Case Execution-time Analysis (WCET Analysis). Despite these demands on the temporal behaviour of hard real-time code, it is still common to operate with traditional algorithms and programming structures which are usually applied to non real-time applications. The objective of these traditional approaches is to obtain high temporal performance for the average case, whereas the worst-case performance is considered to be of less importance. As a matter of fact, however, the WCET of hard real-time code is one of the most important time constants and represents a temporal basic parameter of a real-time system. In my thesis I want to contribute to the insights into an unconventional progamming strategy that supports the construction of code that is well suited for hard real-time systems, swapping the traditional priorities between Average Execution Time (AVG) and WCET. The highly prioritised Worst-Case Execution-time (WCET) is strongly associated with WCET Analysis which involves path analysis of the code which is executed. In order to reduce the complexity of WCET analysis there exists a special progamming paradigm where the key property is to write single-path code. As as consequence, the paradigm makes it possible to obtain a single execution path which makes path analysis and thus WCET analysis trivial. The thesis demonstrates how to apply both, the progamming strategy mentioned above as well as the programming paradigm for single execution paths by exploring several pieces of code, respectively algorithms. Furthermore, the results and comparisons of the different variants of algorithms will be presented and evaluated. The thesis also explores how the concepts mentioned above can be applied

to other classes of algorithms. By doing so it presents new strategies obtaining single-path code with constant execution time for a wide range of programming problems.

Kurzfassung

Harte Echtzeitsysteme gewinnen in unserer Gesellschaft immer mehr an Bedeutung, da sich bis heute bereits eine beträchtliche Anzahl von komplexeren computergesteuerten Systemen, die speziellen sicherheitskritischen Bedingungen unterliegen, etabliert hat. Inzwischen findet man harte Echtzeitsysteme besonders häufig in sicherheitskritischen Bereichen, zu welchen unter anderem der Flugzeug- und Automotivbereich, Atomkraftwerke, Robotertechnik sowie die Militärtechnologie zählen. Der Programmcode solcher harten Echtzeitsysteme unterliegt bestimmten Anforderungen bezüglich der worst-case Performance bzw. der worst-case Ausführungszeit (Worst-Case Execution Time, abk.: WCET) und setzt bestimmte Eigenschaften voraus, um eine effiziente WCET - Analyse zu ermöglichen. Trotz der Anforderungen an das zeitliche Verhalten von Programmcode harter Echtzeitsysteme ist es teilweise nach wie vor üblich, mit traditionellen Algorithmen und Programmstrukturen zu arbeiten, die gewöhnlicherweise in Nicht-Echtzeitsysteme Verwendung finden. Das Ziel dieser traditionellen Ansätze ist vor allem, einen hohen Durchsatz für den Durchschnittsfall zu erzielen, wohingegen die worst-case Performance dabei als nicht so sehr relevant erachtet wird. Dennoch ist die WCET von Programmcode bzw. den Tasks harter Echtzeitsysteme eine der wichtigsten zeitlichen Grössen und macht somit eine elementare Eigenschaft eines Echzeitsystems aus. Hinsichtlich der WCET soll diese Arbeit einen Einblick in eine unkonventionelle Programmierstrategie ermöglichen, welche speziell für harte Echtzeitsysteme geeignet ist. Diese Strategie macht es sich zum Ziel, die traditionellen Prioritäten von Durchschnittsausführungszeit (Average Execution Time, abg.: AVG) und WCET zu vertauschen. Die in diesem Gebiet besonders wichtige WCET ist stark mit der WCET-Analyse verbunden, in welche die Pfadanalyse ebenfalls stark miteinfliesst. Um hierbei die Komplexität der WCET-Analyse reduzieren zu können, existiert daher ein spezielles Programmier-Paradigma mit der Haupteigenschaft, Single-Path Code zu erzeugen. Dieses Paradigma soll ermöglichen, dass das entsprechende Programm auf einem einzigen Ausführungspfad ausgeführt wird. Diese Arbeit demonstriert diesen Ansatz anhand einiger ausgewählter Algorithmen, auf

welche sowohl diese Methode als auch die bereits angesprochene Strategie angewendet wird. Darüber hinaus werden auch Ergebnisse und Vergleiche von den Varianten der entsprechenden Algorithmen präsentiert und die Sinnhaftigheit dieser Vorgehensweise beurteilt. Die Arbeit erforscht unter anderem auch die Ausweitung dieser Konzepte auf andere Klassen von Algorithmen und zeigt dafür auch neue Ansätze für die Vereinheitlichung des Ausführungspfades und der Ausführungszeiten.

Acknowledgements

Foremost, I would like to thank my supervisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner, who gave me the chance to work on this interesting topic and who supported me during this work by sharing his professional experiences and visions with me. I also would like to thank my co-supervisor Dipl.-Ing. Benedikt Huber who gave me very useful input for the whole topic and for my approaches.

I want to thank all my friends who guided and inspired me in my life and supported me during my eduaction. Especially I would like to thank my friends Nedzad, Voin, Ondrej, Yilin and Harald who helped me with difficulties and were always there to share moments of joy.

Lastly, I would like to express my gratitude to my whole family who supported me during my studies in every possible way.

Thank you!

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

In the past years the computer system industry has developed itself strongly in the area of hard real-time computing systems. The demand for this kind of systems emerges every day more and more in our society. Current technologies like automotive electronics or air traffic control but also future developments such as advanced robotics systems rely on specific safety conditions and guarantees related to the functionality of a system which are satisfied by the basic principles of hard real-time systems. One of these principles is to deliver computed results of a task within a certain time interval. In fact this can be achieved if the Worst-Case Execution Time (WCET) of all tasks is known in order to prove that the hard real-time system meets all its deadlines. The WCET is one of the central parameters that characterises the timing of a piece of hard real-time code. Knowing the WCET of a task is a basic requirement for building hard real-time systems. The Worst-Case Execution Time Analysis (WCET Analysis) provides the base for determining WCET bounds for all time-critical tasks of a hard real-time system. WCET analysis assesses the execution time of isolated tasks.

In the past, computer architectures used in embedded hard real-time systems were much simpler in their complexity. Consequently the development of the WCET analysis for this kind of systems shaped up accordingly to the characteristics of these architectures. Timing analysis used hierarchical timing models - models that separate timing issues of the low level task from the real-time scheduling at the high level. By then WCET analysis had started to become an independent field within the research on real-time systems. Methods were developed for identifying paths through pieces of code and strategies were found for computing WCET estimates.

With the further development of computer architectures the WCET analysis had to be extended for being able to compute estimates for code running on more complex architectures. Dealing with the modification and extension

of hardware required to model the effects of instruction pipelining and caches. Although the new hardware came up with lots of new temporal properties the basic timing models of WCET analysis remained unchanged. In order to deal with these changes an additional analysis step was established, using specific hardware and software information in order to achieve a pessimistic estimation of the worst-case effects of the timing interactions between different tasks [13]. Even though there are technologies and tools that allow proper timing analysis of more complex processors systems the effort and the costs of determining the temporal behaviour of such systems are very high. Furthermore the costs of the analysis of such systems are increasing disproportionally with the degree of complexity of the architectures, which clearly shows the gap between the properties of the current hardware and the current capabilities of timing analyses.

However, WCET analysis has not only to deal with characteristics of current computer architectures. It also gets affected by the nature of program code which gets executed on these architectures. The design of program code used in computer systems traditionally has properties which originate from earlier conditions. These properties are representing intuitive programming strategies which are not only inappropriate for the simplifcation of WCET analysis but are also a substantial factor for the complexity of the analysis.

1.1 Motivation

WCET analysis is a field of research which by now has reached its limits in some specific areas, since the effort and the level of complexity needed for analysing a software-hardware construct is not always convenient. Although there are approaches including abstractions and pessimistic assessments it becomes clearer and clearer that for achieving a better performance of WCET analysis it is not only the WCET analysis itself that has to get enhanced and adapted, but rather the domains of hardware respectively software, in other words the entities of these two domains which need to be analyzed by the WCET analysis.

Hardware architectures of the most complex processors are usually not used in embedded real time systems but it has to be said that most processors which are actually used, have possibly problematic features. These kinds of architectures are designed in a way which makes the execution of instruction hardly time-predictable. On the other hand, the software design is another important requirement for the predictability of a computer system and its structure is one of the most important conditions in order to enable an adequate WCET analysis. Besides, there is a strong connection between both of these domains in which

the interdependencies play a crucial role when devoloping hardware or software.

1.2 Approach, Objectives

In order to improve worst-case performance and achieve better prediction of WCET, the WCET analysis has to deal with different areas that include hardware- and software - models. The software aspect includes path analyses in order to identify and explore the possible execution paths through the code. On the hardware level the worst-case timing of possible paths on the target hardware gets modeled, and has to struggle with possibly sophisticated hardware features like instruction pipelines, caches and parallel execution units whose interferences are highly complex and cause timing anomalies. Employing special caches that use using a sophisticated replacement policy can even cause timing anomalies which are not dissolvable.

The purpose of this thesis is to explore the employment of a particular programming paradigm which helps us to reduce the complexity of WCET analysis in the domain of software, more precisely in the field of path analysis. Programming code which is written accordingly to this paradigm only has a single execution path. Thus, the complexity of path anlysis can be reduced enormously by using this approach of single path code. Applying these techniques to some sample algorithms requires a reconception of the particular samples respectively algorithms which is a main purpose of this thesis as well.

1.3 Related Work

The approach of producing code that has a single execution path, denoted as single-path code, is a quite new effort on developing software whose timing is easy to predict and to analyse (in terms of its temporal behavior). Since the realization of this method depends on certain properties of the underlying hardware and most of the common hardware does not satisfy the required conditions, the research field for that area seems to be quite developable.

The theoretical background of that strategy has already been published, see [3], [4] as well as [5]. Also, there exists some work that explores the practical appliance of that approach in combination with appropriate hardware support [6]. Even though most of the algorithmic problems have not beend considered so far, previous work one the single-path approach could already prove that there is huge potential in order to gain considerable advantages regarding the WCET

analyses. This was shown by different experiments on a certain hardware [6].

This thesis is strongly based on the theory of the single-path approach and coincides also with some algorithmic problems that have been evaluated already in earlier publications. The illustration of these algorithmic problems should allow the reader to get a deep understanding of the underlying approach. With this understanding the reader should be able to follow the contributions that are given in this thesis.

1.4 Contribution

The theme of this thesis is the concept of the single-path approach and its extension. These extensions address problems that are part of special classes of algorithms which contain basic arithmetic issues that are not solvable with the original approach. The following summarizes the contributions of the thesis.

- The thesis attempts to identify generalizations for the single-path programming strategy. This is achieved by studying and evaluating several algorithms that are suitable for the single-path approach.
- Using the ARM7 TDMI on a ST730 evaluation board with an associated development environment, the thesis characterises the requirements for the underlying hardware and the compiler.
- By performing experiments with the used target, this thesis illustrates results for the examined algorithms and shows that for all these algorithms a single execution path and exection time can be achieved.
- This thesis presents not only implementations of the single-path approach for already known problems but also shows utilities for that strategy when examining algorithmic problems of a different kind of nature. Therefore, this thesis provides new strategies that use data tables and basic arithmetic functions. These strategies are used to extend the single-path approach in order to gain solutions for mathematical algorithms.
- The thesis concludes with an evaluation and discussion of the results which have been performed during the experiments. Furthermore, the thesis provides proposals for future research in the hardware and the software area.

Chapter 2

Technical Background

2.1 Real-time Systems

A system is said to be a *real-time computer system* if the total correctness of the system behavior does not only depend on its logical result of the computations, but also on the time at which the results are delivered. The real-time computer system is always part of a larger system whereas such a system is called a *real-time system*. A real-time system changes its state as a function of physical time (vgl. Kopetz, Real-Time Systems, S2) [7].

A real-time computing system has to react to certain stimuli, produced by the controlled object, within time intervals which are given by its environment. The instant at which a result must be produced is called a *deadline*.

2.1.1 Hard and Soft Real-time Systms

The classificatin of real-time computer systems primarily depends on the kind of deadlines used in these systems. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If the consequences of missing a firm deadline may result in a disaster, the deadline is called hard. An example for such a scenario would be a train crossing a road with a traffic signal not changing to "red" in time before the train arrives at the crossing, which may lead to a catastrophe.

The design of hard real-time systems is fundamentally different from the design of soft real-time systems. A hard real-time computer system must gurantee a proper temporal behavior of the system under all specified load and fault conditions. In contrast to this a soft real-time computer system is permitted to miss a deadline occasionally.

2.1.2 Tasks of a Real-time System

Usually, a real time system consists of several tasks. A task is the execution of a sequential programm, starting with the input data and the internal state of the task, and terminates with delivering the results and updating the internal state. Such a task is called a task with state. If a task does not have an internal state at its point of invocation it is called a stateless task. A task can contain several functions but generally it is engaged with one particular assignment.

A task without a synchronization point is called a simple task (S-task). Whenever such a task has started, it can continue until its termination point is reached. Because such an S-task cannot be blocked within the body of the task, the execution time of the S-task does not directy depend on the progress of other tasks and thus, can be determined independently. However, it is possible that the execution time of such a S-task gets extended by indirect interactions, for instance by task preemption by a task with higher priority.

A task is called a complex task (C-task) if it contains at least one blocking synchronization statement. This could, be for instance, a semaphore "$wait$" operation. This might happen if the task has to wait until a condition outside of the task has been satisfied or if another task has updated a shared data structure. When using such data structures as a protected shared object, there is only one task which can access the data at any particular moment and therefore other tasks need to be delayed by "$wait$" operations. Hence, the WCET of a complex task is a global issue, since it might depend on the progress of other tasks (vgl. Kopetz, Real-Time Systems, S2) [7].

2.1.3 Time-Triggered and Event-Triggered Real-time Systems

There are two main paradigms for designing a real-time system. Depending on the desired properties of the system it is possible to choose either an event-triggered or a time-triggered approach. In a purely event-triggered system all system activities respectively tasks are initiated dynamically by the occurrence of significant events. On the contrary, in a time triggered-system tasks are activated according to a static schedule. There is great demand for both strategies and since there are different requirments on various real-time systems, both paradigms have their legitmation in our world and may even be combined with each other.

If we consider the transportation system in our society, we would describe private cars as an event-triggered method since they can be used dynamically on demand. On the other hand the public transportation system works in a

time-triggered manner because the usage of buses or trains underlies statically predifined schedules without the possibility for dynamic interactions.

2.1.4 Programming Constraints for Real-time systems

Real-time systems only use a restricted form of programming. This form guarantees that such systems always terminate. Thus they must not contain recursion without explicit bounding. For constructs like loops there are similar constraints concerning their iterations counts. All loops must be bounded.

2.2 Worst-Case Execution Time Analyses of Real-time Systems

There exist several tools and strategies for determining WCET bounds. Some methods are based on static analyses of the program and others use measurement-based approaches. Both, the static and the measurement-basesd methods have to handle certain problems and requirements due to the complexity of hardware and software environment. The following sections will describe these problems and demands and give an introduction to these two categories of approaches and their usage in a more detailed way.

2.2.1 Overview

The worst-case execution time (WCET) is the maximum execution time for a piece of code of a task which performs on a given hardware. We call the shortest execution time best-case execution time (BCET). Depending on which kind of system gets explored, there are different requirements for evaluating the WCET, whereas detailed knowledge of the WCET is important especially for reliable systems, such as hard real-time systems. The worst-case execution time analyses (WCET analyses) computes upper bounds for the WCET.

Naturally the worst-case execution time of a task could be easily given if the worst-case input of the task would be known but usually the worst-case input is neither known nor easy to identify. In generell a real-time system consists of a number of tasks which implement the desired functionality. Such a task comprises different execution paths associated with different execution times, depending on the input data and the environment. Since the state space might be very large, in most cases exploring all possible execution paths would be too

costly. Thus, it is a common strategy to consider just a subset of all possible executions for measuring the end-to-end execution time of a task. However this will overestimate the BCET and underestimate the WCET and therefore does not provide a sufficient guarantee for hard real-time systems. Real bounds on the execution times of a task can be determined only when considering all possible execution times. This can be achieved by using abstraction of the task that helps to obtain a chance for timing analysis, which usually results into a large overestimation for the WCET and an underestimation for the BCET. Besides, it is not uncommon that the WCET estimate is achieved by the knowledge and experience from earlier measurements, though with a large overapproximation added.

Especially the overestimations can be considerably larger than the real WCET of the task. As long as there is room for all tasks in some schedule there is no problem but in the case of an almost full schedule and the request for adding one or more tasks it would be definitely of highest interest to achieve tighter WCET bounds. A task shows usually different execution times depending on the input data and the environment that is given by the state of hardware and software. An example for the typical timing parameters of such a task is shown in Figure 2.1. The upper curve represents the set of all execution times including the subset of executions, represented by the lower curve.

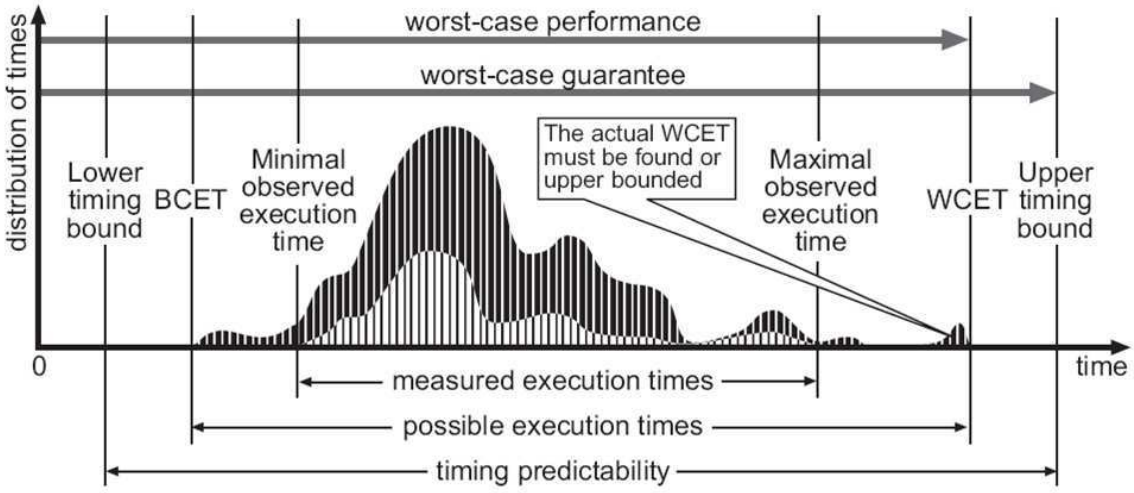

Figure 2.1: Timing diagram for a real-time task [1]

2.2.2 Problems and Requirements

The time for a particular execution depends naturally on the particular path given by a task and the current hardware state. According to the sequence of state-

ments respectively instructions given by this task, which is executed on a specific hardware, there ermerges a particular execution time. The determination of the execution-time bounds has to consider all possible sequences and states with respect to the control flow of the task.

2.2.2.1 Data-Dependent Control Flow

The execution path primarily depends on the initial state and the input of a task. Since this two facts are usually not known, the data structure , the task's *control-flow graph* (CFG) allows to describe a superset of the set of all execution path. The construction of this control-flow graph has to deal with problems created by dynamic jumps and dynamic calls. For instance switch-statements lead to dynamic jumps and cause problems when analyzing machine code since the assembly code sets labels for such branches.

The paths taken through the superset of the CFG depend directly or indirectly on the input data where some paths in the superset will be never taken. These are the ones which have contradictiory consecutive conditons and therefore can be eliminated. Among other things this can be achieved by a phase called control-flow analysis (CFA), also called *High-level Analysis*, which is useful for increasing the precision for subsequent analyses. It includes also the determination of execution frequencies of paths and their relations. Furthmore it has the function of determining bounds of the iterations of loops and on the depth of recursions of routines.

2.2.2.2 Dependences of Execution Times

As already mentioned, timing analyses approaches of the past were used to deal with hierarchical timing models that separate timing issues of the low level task from the real-time scheduling at the high level. In other words, the problem of timing analysis has been solved under the assumption that the timing behavior is context independent. For processors, according to older previous standards, the execution times for individual instruction were independent from the context, that is the execution history, so that times for instructions could be easily looked up in the associated manual of the respective processor. The underlying approach was a structure based one (Shaw 1989) and stated that if a task executes a piece of code A and subsequently another piece B, the total worst-case bound for A;B was the sum of both, the worst-case bound of A and the one of B, formally written as $ub_{A;B} = ub_A + ub_B$

Since the emergence of caches and pipelines in more modern processors

the model of context independence does no longer exist and execution times of instructions may vary depending on the current processor state in which they are executed. That means, if we consider again our example of the task performing two pieces of code, A and B, the execution time of B which is following A, may depend stronlgy on the execution state produced by the execution of the previous piece of code A or even code executed before A. Obviously a structure-based approach is not able to consider this inter-relation.

The *Processor-Behaviour Analysis*, usually called *Low-level Analysis* has to deal with examinations of the processor behaviour in order to model the behavior for a given task and all components influencing the execution times of it. Such influences are caches, memories, pipelines and also branch prediction. Hence, with this analysis it is possible to determine upper bounds on the execution times of instructions.

2.2.2.3 Timing Anomalies

Timing anomalies are an unexpected behavior of different features in modern complex processors and cause problems when performing a WCET analysis on a given target since the involved hardware needs to be integrated into the analysis process as well. The most powerful microprocessors comprise an highly complex processor architecture and therefore suffer from such timing anomalies. The notion of timing anomalies was introduced by Lundqvist and Stenstroem [8]. A timing anomaly is a situation where the local worst case does not implicate the global worst case (vgl. Reinecke et al. 2006).

In the following we want to illustrate the scenario of a timing anomaly by giving some examples. The first one demonstrates the different consequences for having either the occurrence of a cache miss or a cache hit in combination with branch prediction, shown in Figure 2.2

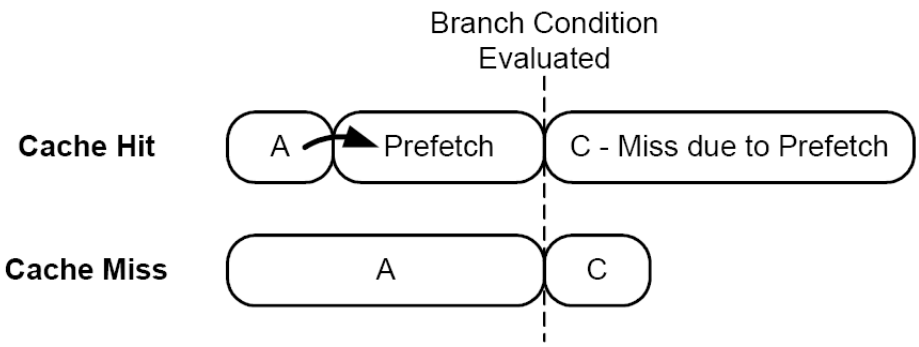

Figure 2.2: Speculation caused timing anomaly [2]

The first case - the case of the cache hit - shows an execution with starting an instruction fetch from the cache, whereas the second case of the cache miss may lead to an execution starting with a cache load and thus possibly causes an increase of the execution time due to the cache-miss penalty. Typically, one would assume to get longer execution times by the occurrence of cache penalties like in the second case. But since the processor speculates on the outcome of conditional branches which results in prefetching of instructions in one of the directions of the conditional branch this assumption is not necessarily true.

Considering our first case now we can have a look at what will happen after finally evaluating the condition and recognizing that the wrong direction has been speculated by the processor before. The consequences are that the cache content gets affected in a negative way and hence all effects have to be undone. Thus, the first case, also observed on the Motorola ColdFire 5307 processor [9], is stated to have a longer execution time than the second one. This illustrates one of the reasons for timing anomalies, specified as *speculation-caused anomalies* (vgl. Reinecke et al.) [2].

Another example for timing anomalies is the *scheduling-caused anomaly*. Because of scheduling characteristics there can arise a scenario where a cache miss (the local worst-case) may result in a shorter execution time, than a cache hit. This is shown in Figure 2.3.

Considering now the second case in the figure, task A obviously benefits from the cache hit and thus takes less time for finishing. But shortening task A leads finally to a longer overall schedule, because task C gets now blocked by task B.

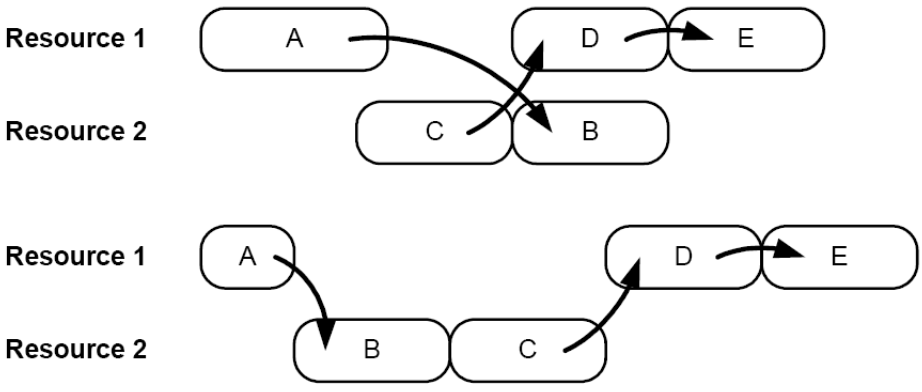

Figure 2.3: Scheduling caused timing anomaly [2]

Summing up we can state that the existence of timing anomalies in a given processor always has a strong influence on the applicability of methods for timing analyses for that processor. The problematic and underlying assumption which is

violated by timing anomalies is basically that in case of a choice always the local worst-case transition can be taken in order to produce the global worst-case execution time. Thus, the consequence and challenge for analysis is that it has to follow executions through several processor states, whenever it encouters states with a non-deterministic choice between successor states. Obvioulsy this may lead to a very large state space which needs to be incorporated into the analysis [9].

2.2.2.4 Methods required for Subtasks of Timing Analysis

There are several methods for the static and measurement-based approaches used for solving the problem of timing analyses. These methods vary from each other and can be automatic, manual or generic. Some of them are applied at analysis time or at tool-construction time. The combination of such methods is required in order to realize a timing analysis method. We want to mention some of the most common ones briefly in the following section.

Static Program Analysis is a generic method for determining properties of the dynamic bahviour of a given task but without executing it. In general such properties are often undecidable. If we consider, for example, the instruction-cache analyses, determining for each program point in the task which instruction will be in the cache at that particular moment, we will recognize that under certain circumstances this may appear to be feasible but usually it is not possible to determine that for tasks whose control flow depends on input data [1]. The problem of input-data dependent control decisions in a task respectively an execution is significant and will be discussed in detail in further sections of this work.

Measurement of a subset of all possible executions can be used as well in order to produce estimates of the WCET. Since this is just about estimates and not bounds, there are no guarentees, thus such measurements may just be usefull for applications like non-hard real-time systems. When considering quite simple architectures it is possible to achieve bounds which are safe enough to obtain guarantees.

Another standard technique for estimating the execution time for tasks on a given hardware architecture is *Simulation*. Although for different types of architecures not all simulators can be trusted as clock-cycle accurate simulators, it is usually possible to achieve rather accurate estimations of the execution time when using an architectual simulator with a sufficiently detailed timing model.

2.2.3 Static Methods

In contrast to measurement-based dynamic methods, static methods are not based on executing code on the real given hardware. These methods include the code itself, possibly with some remarks, and analyze the set of all possible control flow paths through the task. The underlying hardware architecture gets abstracted in order to obtain simplified hardware models. Using them in combination with the control flow allows to compute upper bounds of the execution time. One such approach for static methods is described in detail by Wilhelm [10].

2.2.3.1 Main components of a static timing analysis tool

The following figure 2.4 shows the basis components and the flow information of a static timing analysis tool. We want to discuss the main properties and the functions of the individual parts, with the main focus on *Control-Flow Analysis* and *Processor-Behavior Analysis*.

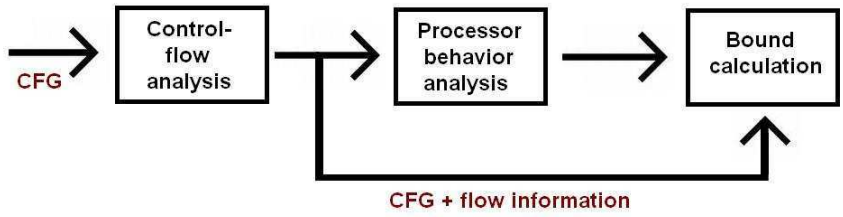

Figure 2.4: Abstract illustration of basic components of a timing analysis tool

The function of control-flow analysis is to collect information about possible execution paths. In general its not possible to determine the exact set of paths. The input of flow analysis consists of a representation of the task which is something like the call graph and the control-flow graph (CFG). There can be also additional information, given by the user, like ranges for input data or iteration bounds on the loops. The result of the flow analysis can be used to define the dynamic behaviour of the task. It provides inforamtion about the fact which function may get called and which dependecies exist between conditionals. In general it provides knowledge about the possibility of paths.

Processor-behavior analysis has to deal with the problems created by typical components of processors that make the execution time context-dependent, such as memory, caches, pipelines and branch prediction. As already mentioned the execution time of an individual instruction may depend on the execution history. Hence, in order to determine bounds for the execution time, it is required

to analyse the whole set of states of processor components associated with the paths leading to the task's instructions. Thus, the whole processor periphery has to be taken into acccount in order to guarantee proper analyses. This includes the full memory hierarchy, the bus and all peripheral units. For this reason a more appropriate term for this analysis would be *hardware-subsystem behavior analysis* [1].

In general the analysis is based on an abstract model of the given processor and all other components including peripherals and naturally the memory subsystem and the buses. The model is contructed in a way that it will never predict the time for an execution to be shorter than the time for the same execution executed on the given processor. The challenge for deriving such an abstract processor model can be quite complex and strongly depends on the category of processor used.

2.2.4 Dynamic (Measurement-based) Methods

These methods are solving the problem of timing analysis by executing the given task on the given hardware or on a appropriate simulator. Then the execution time for the whole task or some parts of it is measured by considering some set of inputs. Many different measurements have to be done on many different inputs and the highest time measured might be the actual WCET.

Since the set of inputs does not necessarily cover all possible input-sets by which not all possible combination of inputs can be measured, there is no guarantee that the actual WCET has been found. In other words, if the subset is not guaranteed to contain the worst case, measurements of this subset of all possible executions can produce only estimates for the execution time but no bounds. Thus, there is always a safety margin added to the measured WCET, when using such a method [1].

There are several approaches to measure the execution times of code segments which are typically CFG basic blocks. The results of these measurements are getting analysed and finally can be used for bound calculation in order to achieve estimates of the WCET and BCET. Thus, such measurements can be used to replace the processor-behaviour analyses used in static methods and the path-subset problem can be solved by using control flow analysis to find all possible paths. Subsequently overall time bounds are computed by combining the measured times of code segments which is done by bound calculation.

Although this approach would consider all possible paths it could still produce unsafe results because of the fact that only a subset of possible contexts (initial

processor states) is provided for each basic block respectively code segment. In fact it is possible to decrease the risk of unsafe results by running some additional tests to measure more contexts but usually testing of all execution paths is probably not possible and usually unworkable. Thus, we can summarize that when using measurment methods, determining the worst-case initial states is still hard or almost impossible for complex processors and only for processors with simple timing behaviour it is feasible to compute bounds on the execution time.

2.2.5 Comparison of Static and Measurement-based Methods

Static methods are able to compute bounds on the execution time by using control-flow analysis and bound calculations to cover all possible execution paths. Abstraction is used to cover all possible context dependencies in the processor behaviour. In order to obtain this level of safety an enormous effort for producing processor models has to be done. Furthermore, there is no guarantee that the obtained results for the bounds of WCET are precise. Thus this can results in an overestimation. On the other hand an enormous advantage of static methods is the fact that the analysis of a particular program can be done without running the program itself. Thus, costs for complex hardware and simulation of the hardware and peripherals of the given system can be saved.

Measurement-based methods can replace processor-behaviour analysis by measurements. The measurment-based methods usually use control-flow analysis to include all possible execution paths in order to perform an estimate calculation. The safety of the measurement-based method strongly depends on the fact whether it is feasible to measure all existing execution paths. Another aspect regarding the safety is the fact that measurements have to get started in a worst-case initial state under the condition that the processor is simple enough to do so. But as already mentioned in previous section, this is not always achievable.

However, one major advantage claimed for measurement-based methods is the fact that they are simpler to apply to new target processors, because modeling of the processor behaviour, which is highly expensive, then becomes obsolete. Another advantage is that these methods usually produce more precise estimates for the WCET and BCET which are closer to real timing values than the bounds obtained from static methods, especially for complex processors.

Therefore, measurement-based methods can be used for validation of static analysis approaches. With the results of measurement-based analysis it is pos-

sible to obtain a picture of the variability of the execution times of the application. Thus, comparisons of the results of both, static and measurement-based methods, is possible and may lead to a more reliable rating of the results achieved by the static analyses. For instance, measurement-based methods should not deliver execution times that are much lower than the ones predicted by static analytical methods. If there occurs a remarkable difference between the delivered execution times this would indicate that the values of the static method are inaccurate [1].

2.2.6 Limitations of WCET Analysis

From a theoretical point of view, static WCET analysis has some advantages over the measurement-based approach. As already mentioned, the guarantee for determining reliable bounds on the WCET cannot be ensured by using solely measurement-based techniques. Furthermore, if only partial knowledge of the control flow is available, calculating the WCET bound with static analysis results in a safe bound as well. Both methods, the static and measurement-based one, have their limitations due to several reasons.

In practice, one of the limitations of static WCET analysis concerns the flow facts which describe the possible control flow paths of a program. Generally, determining flow facts is not just an automatic process but rather requires code inspection and code annotations by the programmer in order to achieve precise control flow paths. This information is then used to calculate bounds on the WCET. However, in practice such specifications about flow facts are not powerful enough to cover the requirements for several architectures. This may be possible for simple processors without caches and pipelines but for modern processors this information is not sufficient. This is because of the fact that the characteristics of pipelines and caches depend strongly on the concrete execution order of instructions. Hence, determining the flow facts need to incorporate the execution order of instructions in a much more detailed way which is even more complex and might lead to additional pessimism.

At the same time we have measurement-based approaches which perform measurements on the runtime and do not depend on information concerning the control flow. Nevertheless, obtaining WCET bounds requires to determine all relevant execution scenarios for beeing measured. Such an execution scenario is given by an initial state of the target hardware and a certain set of one or more parameters representing the input data. The remaining problem is now to make sure that all relevant values for the input data have been found, because only then

it can be ensured that all relevant execution scenarios were tested. Considering the whole value space of possible input data is generally not feasible because of the costs and efforts. However, the miss of considering some relevant values for the input data can result into an underestimation of the WCET bound. In order to avoid this problem the WCET bound is usually extended by some additional offset.

Hence, we can conclude that the limitations of measurement-based approaches and static WCET analysis are somehow analog. Static WCET analysis methods require sufficient flow facts in order to model the control flow of a given code where measurement-based methods need to provide precise information about execution scenarios to be tested. The requirements of both of them cannot always be satisfied, thus pessimistic estimations are the consequences.

2.2.6.1 Hardware Reality and Analysis Reality

As already mentioned in Section 1.2 the WCET analysis is not only limited by the magnitude and diversity of written software but also refuses to work when trying to model the timing behavior of some sophisticated hardware. The situation for timing analysis capabilities versus complexity of current hardware, introduced by Kadlec and Kirner [11], is illustrated in Table 2.1

It shows us the wide gap between current properties of hardware and the current capabilities of timing analysis. Hence, CPUs including features like a Pseudo Least-Recently-Used (PLRU) cache replacement strategy are not suitable for embedded real time systems since they are not analysable and can cause unbounded effects on the WCET.

	timing analysis capability	current hardware
Caches		
levels	level 1	level 3
separation	separated (Harvard)	2nd and 3rd level combined
associativity	2-way	up to 16-way
replacement	LRU	PLRU, PRR
Branch Prediction		
history	local 2-bit saturating	mixed local / global
locality	local only	mixed local / global / tournament

Table 2.1: Analysis Capabilities and Current Hardware

Considering Moore's law, in the last centuries the transistor count per chip has doubled about every 2 years due to hardware improvements and thus also has been improved the performance. On the other hand we have Proebsting's

law, stating that the performance doubling due to compiler technology happens only every 18 years.

Both "laws" are criticized for their extrapolation but their disputed statements together express the industrial reality of the past thirty years.

As stated in [11], the complexity of timing analysis is rising at least as fast as the complexity of compilers and thus it is highly unlikely, that analysis precision can catch up with the imprecision gained by already present hardware features.

2.3 Processor Architecture and Compiler

There are some general properties and characterisitics when dealing with architectures and compilers. Since we have to consider several points of those issues in further sections, including measurements of execution time but also compiler-behavior, we want to highlight some common terms and concepts, relevant for further treatment in this thesis.

2.3.1 Common Terms

The term (computer) architecture includes the following three main subcategories, covering the three aspects of computer design [12].

- Instruction Set Architecture (ISA): The ISA is the abstract image of the computing system from the view of the machine language (assembly language), including the instruction set, word size, adress modes and process registers as well as adress and data formats.
- Microarchitecture - also known as Computer organization: This represents a more detailed lower-level description and describes how the individual parts of the system are interconnected and how they interact with each other in order to implement the ISA. For instance, the size of the cache would be such a property and thus is an organizational issue.
- System Design (Physical Hardware): This includes all other hardware components of a computing system such as computer buses, memory controllers, hierachies and other issues like multi-processing.

In this work we are mainly dealing with the ISA which defines an abstract interface between hardware and low-level software. For a specific ISA there might be different implementations, which might have varying costs and performance

but in any case, the functional requirements of an ISA have to be met by the implementation.

There are rising up some important facts for WCET analysis when considering the separation of hardware architecture and implementation. A routine respectively program, which needs to be analyzed, may exist in different forms, depending on the compilation process, beginning with the high-level language (source File in C code), subsequently assembly code, after this the object file and finally an executable binary.

These different forms are identified as representation levels [13]. Even though the covered information satisfies the characteristics of the ISA, the timing information necessary for WCET analysis is not available. Hence, it is very important to consider the complete architecture for WCET analysis, since the whole timing information can be obtained only when considering both, the hardware- and the software design.

2.3.2 Execution Time

Since we are going to determine the execution times, when treating several experiments in further sections, we want to highlight some metrics. In order to analyze execution times for programs it is generally possible to use one of the following definitions. According to [14], the total execution time of a program p can be defined as

$$CPU\ time = CPU\ clock\ cycles_{Programm\ p} * Clock\ cycle\ time \tag{2.3.1}$$

with

$$Clock\ cycle\ time = \frac{1}{Clock\ frequency} \tag{2.3.2}$$

Regarding hardware performance, it can be sometimes meaningful to focus on the total number of executed instructions (IC). To reason about the performance, the term $cycles\ per\ instruction$ (CPI) is given by

$$CPI = \frac{CPU\ clock\ cycles_{Programm\ p}}{IC} \tag{2.3.3}$$

wheras the CPU time of the programm can be expressed as

$$CPU\ time = IC * CPI * Clock\ cycle\ time \tag{2.3.4}$$

The $instruction\ count$ (IC) is determined by the compiler technology which is

used and moreover by the ISA. The $cycles\ per\ instruction$ (CPI) depend on the organization and instruction set of the architecture, wheareas the clock cycle time depends on the hardware technology.

For usual experiments like shown in further sections of this thesis, it would be sufficient to restrict measurments regarding the execution time of a given program on the number of $CPU\ clock\ cycles$ required by the that program.

2.3.3 Compiler Issues

Compilers translate high-level programming languages (such as C) into assembly code for a given target processor. In the area of processor-based embedded systems, the usual employment of well-established compilers is neither common nor always possible. Under certain conditions, program designers need to adapt the outcome of compilers and sometimes they are even forced to use only assembly language to programm the desired features for an embedded application.

Although the programming effort of assembly compared with C is quite huge and yields problems such as restricted code portability and even a reduced dependability, such a programming approach has still enough eligibility when designing embedded systems. This is because of the fact that embedded systems very often require an extra high performance and efficency that can be provided by an adequate machine code.

But another fundamental reason, arising in this thesis, lies in the fact that widely-used compilers deform the potentially desired structure of the high-level routine, with respect to the control-flow of that routine. Basically this happens due to several optimizations performed by the compiler, in order to speed up the final code for the target processor.

Thus, the choice of an appropriate compiler for a given target hardware is essential when trying to maintain specific properties whereas the beneficial effect will get clear and more traceable, when exploring further sections of this thesis.

Chapter 3

Approach

3.1 Problem

The problem of achieving predictable code originates basically from the intuitive requirements of developments in computer science in the past where the basis for elementary properties of software and hardware were laid - properties, which, in certain areas, have still been predominant until today. With the appearance of software relevant to safety, hard real-time systems came up in order to satisfy the requirements given by these applications in these days.

In order to facilitate the problem of predictability and thus WCET - analysis, program code for hard real-time systems has to satisfy several conditions. Apart from the already mentioned properties of programming style, including restrictions for recursions and loops, there are some further characteristics which have to be satisfied in order to generate code that is well-suited for hard real-time systems. These characteristics - illustrated by a special programming paradigm, respectively strategy - are going to be explained in detail in the following sections.

3.2 Requirements of Real-time and non Real-time Programming

Requirements imposed on real-time and non real-time systems differ in several ways. For now we want to compare the different performance requirements regarding the program code for these systems.

If we consider systems which do not operate in a time-critical context, high throughput would be the main performance goal. Naturally, this can be achieved if the execution times of the main part of a certain task are short - in other

words: speed optimization for the most probable execution scenarios is required in order to achieve a better performance for the average case.

In contrast, the average execution time of the task in hard real-time systems is less important. Instead, it is essential that any time-critial task meets its deadline under all circumstances, even under peak load (Kopetz) [7]. In order to ensure this, real-time tasks need to have a short WCET.

Formally the different performance requirements are described by Puschner [4]. It is assumed to write a code that solves a particular problem. The space of possible input data for this problem is $DELTA$ = {δ_i} on a machine μ. The set P = {p_i} denotes the probabilities with which each of the input-data scenarios occurs. For example, δ_i would be such a scenario.

According to [4] it is further assumed that PI= {π_i} represents the (infinite) set of correct code samples which is able to solve the given problem. The function c = c(π_j, δ_i, μ) returns the execution time of π_j with input data δ_i on the machine μ. Considering this set of definitions there are now emerging two totally different formulations given by the following lines.
The following criteria derived in [4] identify the most appropriate solution for real-time, that is π^{μ}_{wcet}, respectively non real-time programming code, which is π^{μ}_{acet}.

$$\pi^{\mu}_{avg} \in \{ \pi_s \mid \forall \pi_j \in \Pi : \sum_{\delta_i \in \Delta} p_i c(\pi_s, \delta_i, \mu) \leq \sum_{\delta_i \in \Delta} p_i c(\pi_j, \delta_i, \mu) \} \qquad (1)$$

$$\pi^{\mu}_{wcet} \in \{ \pi_s \mid \forall \pi_j \in \Pi : \max_{\delta_i \in \Delta} c(\pi_s, \delta_i, \mu) \leq \max_{\delta_i \in \Delta} c(\pi_j, \delta_i, \mu) \} \qquad (2)$$

Comparing the two formulas clearly shows the different intentions for real-time and non real-time programming. The selection criterion in (1) for the most appropriate solution does not really consider execution times of individual input-data scenarios. Only scenarios with high probability have a strong influence on the chosen solution π^{μ}_{acet}.

In contrast to (1) the selection for (2) happens in a way that all scenarios - independent of their probability of occourrence - are treated equally. Thus, even the scenario with the smallest probability may influence the choice for the program.

3.2.1 Traditional Performance-Oriented Approach

The typical program design for non real-time systems usually provides a speed optimization for the most frequently used scenarios. The techique to implement

such an optimized behaviour for specific frequent cases is to use program code which processes its actions to be performed on the basis of input data. That means that optimization for good performance is based on using input-data dependent control decisions. Furthermore, this is the only effective and straight way to achieve short execution times for the favoured input-data sets. Therefore, such a programming style influences the quality of the achievable WCET in a negative way. First of all, the non-favoured inputs need to be tested as well, but since all execution times have to be considered in an equal way, the advantages of favoured input data regarding their execution times cannot be exploited. Another important point is that this might result in different computational costs. This is because of the fact that overall complexity gets distributed unevenly over the input-data scenarios. In other words: a cost reduction for some part of the input-data space will cause higher costs for the rest of the inputs.

Intuitively we would assume that if an implementation achieves short execution times for common scenarios (favoured input-data sets), we also suppose that, in return, rare scenarios (non favoured input-data sets) would cause higher execution times.

3.2.2 Worst Case - Oriented Method

In contrast to traditional performance-oriented methods the approach for worst case programming uses completely different strategies. The strategy discussed in the section before, where data-dependent control decisions are used, is based on intuitive optimization patterns which we use in everyday life. On the other hand WCET oriented programming is neither intuitive nor easy to model. It needs a unusual way of thinking, one which is quite different from the solution strategies normally used. Thus, code pieces respectively algorithms produced according to this new approach very often do not appear to be straightforward.

In order to avoid certain shortcomings of the traditional approach the necessity of new programming strategy is given and formulated by Puschner [4]. It proposes WCET-oriented programming that aims at writing code that is free from input-data dependent control flow decisions and if this cannot be completely achieved, operations which are only executed for a subset of the input-data space should be restricted to a minimum.

The outcome of applying the novel programming strategy should result in pieces of code having no (or just a few) input data depending branches or loop-conditions and moreover they should have a better or adequate WCET. Unfortunately, it is not always possible to treat all inputs identically and also the WCET

might be higher than in the traditional approach. Basically, this depends on the kind of algorithm used.

Another important point, which has to be made, is that the way of programming has a huge impact on the complexity of WCET analysis. There is a strong relation between the number of input-data-dependent alternatives and the number of execution paths through a piece of code. The appliance of this fact can be exploited when one tries to determine the WCET for a given piece of code, because evaluating a smaller number of paths for WCET analysis is much easier, especially when working on an sophisticated architecture. Besides, the chance of faults is much smaller when characterising all the paths.

3.3 The Single Path Approach

Programs usually behave differently for different input data, that means that depending on the input, code is executed on different execution paths. Hence, the central idea of the single-path approach is that programmers design programs, respectively algorithms the behaviour of which is independent of input data and which can be executed on one single execution path. Having a single execution path would make WCET analysis very easy.

Such an approach may seem to be quite restrictive and one would think that programmers would be just allowed to write entire simple programs, but it can be shown that the single-path approach is not that restrictive at all. This belief is justified by the fact that any piece of code that is WCET-analysable can be translated into code with a single execution path [3]. Such a translation uses if-conversions [5] in order to produce code that keeps input-data dependent alternatives in the code local to single conditional operations with data-independent execution times [3].

3.3.1 The Constant-Time Conditional Expression

In order to transform input-data dependent branches and their alternatives into sequential code and thus achieving predictable program code, a special feature is needed for implementing the desired behaviour: it is the *constant-time conditional expression* operation. A constant-time conditional expression consists of a boolean expression *Cond* and two expressions *Expr1, Expr2*. For this expression we use the following notation, as introduced in [3].

Depending on the boolean condition *Cond* either the value of the result of the first expression *Expr1* or of the second one *Expr2* is taken. More precisely, if

Cond evaluates to *true* then the result of *Expr1* is the value of the constant-time expression but if *Cond* evaluates to *false* then the value of the constant-time expression is the result of *Expr2*.

Cond # Expr1 : Expr2

Choosing the syntax of the constant-time conditional expression like described above had the following reason. The well-known conditional assignment operator (cond ? expr1 : expr2) used in the C programming language has quite similar intentions and if we assume that *Expr1* and *Expr2* do not have any side effects, the result would be actually the same.

However, if we have a closer look on both of the expressions, we will discover some elementary differences. The standard C conditional expression represents the classical *if-then-else* - construct. It evaluates the condition at first and, depending on the result, it subsequently executes either the branch for *true* or the one for *false*. Unlike the C conditional expression, the constant-time conditional expression evaluates both expressions at the beginning and evaluates the condition afterwards. Finally, the only step left to be done - depending on the condition evaluated before - is to return one of the two expression results as the value of the whole operation. The different semantics of the two conditional operators are shown in Figure 3.1.

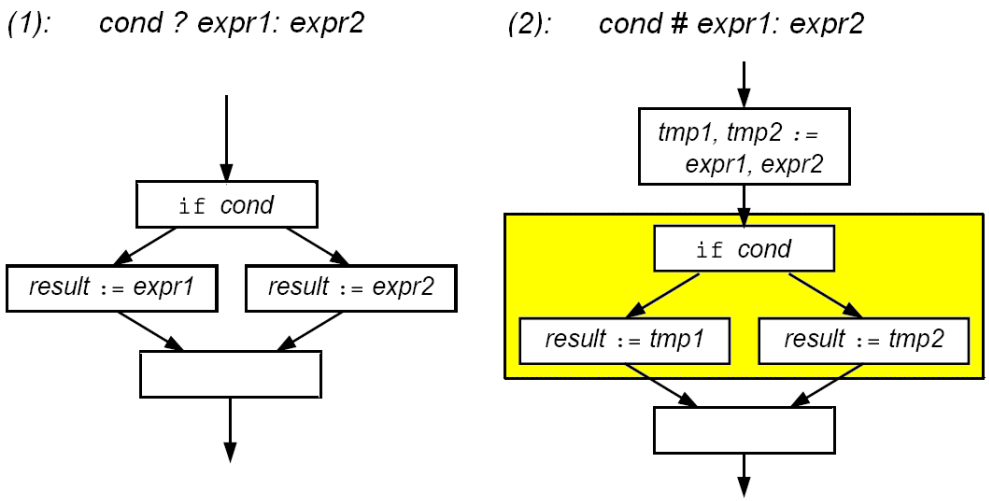

Figure 3.1: Standard C conditional (1) and constant-time conditional (2)

Since the C conditional expression (1) only evaluates one expression, the constant-time conditional expression (2) obviously takes more time to perform

than the C conditional, and if we assume, for instance, that both expressions inside would require equal effort, the time needed by the C conditional would be twice as much. Nevertheless, the advantage of the constant-time conditional should become clear if we consider for instance two totally different alternatives of a branch. In the C conditional (1) these alternatives would lead to different execution times, whereas this problem cannot occur when using a constant-time conditional expression (2).

The latter works as follows: Both alternative expressions are executed in a sequence. Then, the result of the overall conditional is selected and finally gets returned within constant execution time (see the bottom of the box at (2) in 3.1).

Generally, it is not possible to predict the exact execution time of a construct such as given in (1) and in contrast the particular power of (2) is based on the fact that it has always a single execution time. By the unconditionally execution of both expressions, the problem of determining the worst-case behaviour of the conditional becomes superflous. Thus, this approach offers exactly what is needed for an adequate and easy code prediction.

3.3.1.1 Issues on the Constant-Time Conditional Expression

For the implementation of a correct constant-time conditional expression there are still certain challenges that have to be overcome when trying to build such a construct that ensures that the conditions, explained in the section before, hold for any case. The attention here lies particularly on the branch contained in the constant-time conditional operator, which occurs right before the result of the overall conditional is returned. Such a branch is of great importance due to the following reasons:
Even if the operations of the two alternatives in the constant-time conditional expression would be equal, these two alternatives could result in different execution times due to the remaining conditional branching statement which still needs to select the final result. This may happen because of the following reasons, described in [3]:

- First of all, the time needed for the branching itself may differ, depending on which alternative was selected. For instance, in a piplelined CPU, the pipeline may stall on a jump when executing the first alternative while no stall occurs if the second one gets executed.

- Furthermore, there is the problem with computers having an instruction cache. The executions of the two alternative instructions of a branch may leave the cache in different states. Since different cache states potentially

influence the execution times of instructions, an exact prediction of the execution time becomes impossible.

In fact, the first problem could be solved by stuffing the shorter alternative for instance with NOP instructions in order to obtain both alternatives with an equal execution time. In order to achieve this, detailed knowledge about the timing of the target system would be required, which is not always provided by the environment. Since the second problem also remains unsolved, the implementation of the correct constant-time conditional expression does not seem to be as simple as assumed.

3.3.2 The Conditional Move Instruction

The solution for overcoming these problems has to be established on a basic level and is proposed in [3]. This solution avoids input-data dependent branches at the instruction level and uses if-conversion [5] in order to translate input-data dependent branches into sequential code. This is realized by using the *conditional move instruction* which is implemented on almost every modern processor (Motorola M-Core, Pentium P6, Alpha and some more). With this instruction, an implementation of the constant-time conditional expression that satisfies the conditions explained in the section before, is achievable.

The conditional move instruction has the following form:

---

movCC destination, source

The conditional move compares the condition code CC with the condition code register. If it evaluates to true, the processor copies the contents of the *source* register to the *destination* register. If it evaluates to false, the value of the *destination* remains unchanged.

3.3.3 Converting Code into Single-Path Code

In the following sections we want to give a brief overview of a method which applies this instruction in general in order to achieve single-path code. Furthermore, we want to illustrate some basic techniques in order to bypass the issues regarding single-path code of usual programming constructs.

3.3.3.1 Converting Conditional Code

As already mentioned several processors are able to implement the constant time conditional operator by using conditional move instructions contained in their instruction set, with the objective of performing an if-conversion. A piece of code should show the implementation for the ARM7_TDMI processor [15] which is also used for our experiments, treated in further sections. We want to note at this point that there are different implementations possible since every hardware might have a sligthly different conditional move instruction set, but the basic concept remains the same. The pseudo-code of both, the high-level and low-level language, is shown in the following code Listing 3.1.

Listing 3.1: Branching statement with sequential code of an if-conversion

```
                                    temp1 := exp1;
if cond                             temp2 := exp2;
then result := exp1;                cmp cond 0;
else result := exp2;                movgt result temp1;
                                    moveq result temp2;
```

The left side of the illustration shows the usual branching statement of an if-condition. On the right side, the pseudo-code of the instruction level for performing a translation into single-path code is given. At first, the expressions *exp1* and *exp2* are evaluated. Both results are stored in the registers *temp1* respectively *temp2*. Subsequently, the condition *cond* is evaluated and used within a compare instruction. Depending on the variable *cond* the compare instruction is either stated to be $equal$ (in the case of *cond* = 0) or "greater than" (in the case of *cond* = 1). Thus, exactly one of the two following movCC instructions can get executed and the register $result$ finally holds the intended value, which is either the content of *temp1* or *temp2*.

The implementation of such a conditional assignment has no branches. Hence, this makes it possible to translate any conditional code into a sequential form. The code shown on the right side of the figure above has a single and thus predictable execution time.

Typically, there might be more than one variable contained in the program code which needs to be assigned to different values depending on the condition before. Therefore, we have to take into account every input variable of the two alternatives of the branch. According to [16], we can describe this more formally by assuming that each branch uses the input variables v_1‘,...,v_m‘ in order to compute the values for the final variables $v_1, ...v_n$. This is illustrated in the following parallel code of Listing 3.2

Listing 3.2: General translation of branching statements - *if - conversion*

```
                                    (t1 ,..,tn):= A1(v1' ,..,vm')
if cond                             (t1' ,..,tn'):= A2(v1' ,..,vm')
then (v1 ,..,vn):= A1(v1' ,..,vm')  cond:
else (v1 ,..,vn):= A2(v1' ,..,vm')  (v1 ,..,vn):=(t1 ,..,tn)
                                    not cond:
                                    (v1 ,..,vn):=(t1' ,..,tn')
```

In the first two lines of the code on the right side we see the two assignments resulting from the computation of the input values v_1‘,...,v_m‘. For both alternatives of the branch, the results of this computation or rather the evaluation are stored in tupels of temporary variables t_1‘,...,t_n‘ respectively t_1,...,t_n. The last two lines are assignments of tupels, representing a number of conditional move operations. Depending on the condition *cond* the conditional move operations are used to assign either the one or the other tupel of computed results which are stored in the temporary variables, to the final variables $v_1, ...v_n$.

3.3.3.2 Converting Nested Conditional Code

Another requirement which is necessary in order to achieve sequential code is to get rid of nested if-statements. The main idea is to combine the enclosing condition with the nested condition inside. For this kind of translation, attention has to be paid to the fact that the enclosing condition must not get violated by potential statements before reaching the nested condition. This can be achieved by temporarily storing the value of the enclosing condition so that it can be reused if the value has changed. After doing the translation, the resulting entire condition can be translated into sequential code by applying the principles of if-conversion as described above. The following comparison in Listing 3.3 shows how such statements can be translated.

Listing 3.3: Translation of nested *if* statements

```
..                                  ..
...                                 ...
if cond_A        {                  if cond_A
   then [body C]                       then [body C]
   if cond_B
      then [body D]                 if cond_A and cond_B
}                                     then [body D]
...                                 ...
..                                  ..
```

3.3.3.3 Translation of Loops

The usage of loops is another familiar construct which helps to implement the functionality of the desired behavior of a program or an algorithm. Usually loops are implemented in a way that the number of iterations depends on the given input-data and thus may vary highly.

Naturally, the way to avoid variations of the number of iterations would be to change the loop into a simple counting loop with a constant iteration count. This is done by setting the iteration count of the new loop to the maximum interation count of the original loop. Furthermore, the termination condition of the old loop is used to place a new branching statement inside, with the objective to ensure the termination of the new loop. This is done by introducing a new variable *end* which is set before the loop body. This variable stores the information about the fact if the original loop would still execute the loop body or would have already terminated.

Finally, the entire loop can perform in constant time since the generated conditional statement gets transformed into a constant-time conditional assignment. This is shown in the following Listing 3.4 where the original loop and the new loop are translated to single-path code.

Listing 3.4: Translation of Loops

```
                                    end = false;
                                    for i:=1 to k do{
while cond do max k times{            if not cond
        [body of stmts];                then end = true;
}                                     if not end
                                        then [body of stmts];
                                    }
```

For the occurrence of nested loops which are enclosed by some if-condition we can apply the same strategy as described already in the section before. This consists in generating the branching statement by combining the conditions for termination of the loop and the one emerging from outside. The following Listing 3.5 shows the translation of a combination of an if-condition with the nested loop inside.

Listing 3.5: Translation of Loops and Nested Conditions

```
                                        end = false;
if cond_A{                              for i:=1 to k do{
   ..                                     if not cond_B
   while cond_B do max k times{              then end = true;
        [body of stmts];                  if cond_A and not end
   }                                        then [body of stmts];
}                                       }
```

3.3.3.4 Applicability of Conversions

The transformations as explained in the Sections 3.3.3.1 to Section 3.3.3.3 are not always suitable due to the following reason. When applying the described transformations to real-time code, the conversions of if-conditions and loops may yield temporal predictability at a very high cost in terms of execution time. Thus, this methodology is more a demonstration of the transformation than a generally applicable approach and can be only used if the emerging costs are arguable, which depends on the nature of the program respectively the algorithm.

Summing up, we can state that it is not our intention to simply use this kind of transformations in order to generate temporally predictable code from arbitrary real-time programs. It is rather more essential to generate new algorithms with no or minimal input-data dependent branches, in order to satisfy both, the requirement of temporal predictability as well as an adequate performance. Only when using such new optimized program code, the systematic use of such transformations would make sense and might help to overcome the remaining issues causing the undesired unpredictability.

3.3.4 Exemplification for comparison-based Algorithms

In the following section we want to outline, how the theory introduced in the Sections 3.3.1 to Section 3.3.3.4 can be applied on building new single-path oriented algorithms. Therefore, we are going to provide several examples which should illustrate the main differences between both, the traditional and the WCET-oriented alternatives.

3.3.4.1 Remarks on Instructions and the Compiler

- First of all, we want to note that for the algorithms in the following sections we assume that the effect of the standard conditional assignment operator equals the behavior of the constant-time conditional expression as introduced in section 3.3.1. In order to keep this in our mind, we use the already introduced *"#"* operator instead of the standard C conditional operator (*"?"*). Despite the fact it is rather a model than a real implementation it should basically help us to illustrate the notion of the algorithms.

- Furthermore, we want to advise the reader to pay attention to another crucial assumptions which are strongly required in order to implement the desired behaviour of the upcoming algorithms. Since the instruction set of the used hardware is not always identical and since the behaviour of compilers may vary due to optimization patterns , it cannot be assured that algorithms, after being compiled, will maintain their single-path character as well as their uniform execution time.

  Since our single-path approach is based on the fact that instructions can be executed conditionally, we want to point out two different examples which depend on the underlying target hardware (processor). If we assume that the conditional move instruction does not perform in constant time since its execution time depends on the state of the conditional flag, we would need a compiler which translates the source code in a way so that both properties, the single execution path as well as the constant execution time, can be achieved.

  The following pseudo code, adapted from the Listing 3.1 in Section 3.3.3.1, shows such an example, where individual parts might be dropped by compiler optimizations and therefore might cause different execution times due to a non-constant conditional move instruction. This is because the conditional move instruction produces different execution times which depends

on the fact if the conditional flag of the previous compare condition has been set either to *true* or *false*. is

Listing 3.6: Branching statement illustrating the code(1)

```
                                    temp1 := exp1;
if cond                             (temp2 := result;)
then result := exp1;                cmp cond 0;
else result := result;              movneq result temp1;
                                    (moveq result temp2;)
```

Hence, there is not only the request for an suitable instruction set, supported by the target architecture, but also the need for an appropriate compiler which is able to maintain the desired properties.

The pseudo code given in the following Listing 3.7 demonstrates how the same source code can be translated to low-level code if the conditional move instructions of the processor has a constant execution time. In the first of the two possible cases the instruction executes normally if the conditional flag of the compare instruction has been set to *true*. In the other case the conditional flag has been set to false and the conditional move is just executed as a NOP (no operation). At the ARM7 processor, which will be also used for our experiments in Section 4.3, both options of the execution always need one single computation cycle.

Listing 3.7: Branching statement illustrating the code(2)

```
if cond                             temp1 := exp1;
then result := exp1;                cmp cond 0;
                                    movgt result temp1;
```

The last example represents the main property for our single-path approach and will also show up in similar form when presenting the upcoming algorithms and their low-level code in Section A.1.

3.3.4.2 FindFirst Algorithm

For the beginning, we want to have a look on a very basic algorithm called *FindFirst*. The task of this algorithm is to return the index of the first occurrence of a given key value. The list of values is unsorted and given by an array. The

traditional method of this algorithm, shown in the Listing 3.8, starts from the first element of the given array and compares then the given key to each element contained in the list. As soon as a match between the compared elements occurs, the algorithm returns the current array index and terminates.

Listing 3.8: Traditional Version of FindFirst

```
1  static int findfirst1(int *keys, int size, int key)
2  {
3    int i;
4    int position = -1;
5    for(i=0; i<=size-1; i++) /* max. iterations: SIZE */
6    {
7      if (keys[i] == key)
8      {
9       position = i;
10      break;
11     }
12   }
13   return position;
14 }
```

The WCET-oriented variant of the *FindFirst* algorithm, shown in the Listing 3.9, starts from the last element and traverses, unlike the traditional algorithm, the complete array. Whenever it encounters an element which equals the given key, the index of this element gets stored in a variable. Because of the fact that the searched key can be contained in the array more than once, the variable potentially gets overwritten several times. Since the direction of traversing happens backwards, the returned index is indeed the index of the first element which matches the key.

Listing 3.9: WCET-oriented Version of FindFirst

```
1  static int findfirst2(int *keys, int size, int key)
2  {
3    int i;
4    int position = -1;
5    for(i=size-1; i>=0; i--) /* max. iterations: SIZE */
6      {
7       position = ((keys[i] == key) # i : position);
8      }
9    return position;
10 }
```

Even though this example is quite simple, it perfectly shows the main steps required in order to achieve single-path code. Since we will go through several examples for adaptions of algorithms we want to accent the observed properties, which we simply derive by the given algorithm.

Sequence of required steps in order to achieve WCET-oriented respectively single-path code:

- First of all, the idea of the algorithm has to be changed (that is, traversing backwards with overwriting).
- As a consequence, the number of iterations of the loop can be fixed to a constant value (the maximum) and the *break* is removed out of the loop body.
- Finally, the if-conversion was applied and hence the if-statement has been replaced by a conditional expression. The latter one is a key property for achieving single-path code and ensures in that case that either the variable *position* gets replaced by some new value contained in variable *i* or that it just gets overwritten by itself.

  The method of unintuitive overwriting of variables is an important key feature when modifying certain algorithms and will also guide us through further single-path code examples.

3.3.4.3 Bubble Sort Algorithm

Another example showing how the conditional assignment can be used in order to achieve single-path code with predictable execution time is given by a modification of the well-known *Bubble Sort* algorithm.

Bubble Sort is a very basic sorting algorithm which steps through the list to be sorted, comparing each pair of adjacent items and swapping them if they are not in the correct order. This is repeated until all possible comparisons are evaluated and therefore all required swaps are performed. Finally the list is sorted.

Bubble Sort has both worst-case and average complexity $O(n^2)$, whereas n is the number of items being sorted. Although there exist other sorting algorithms with better worst-case or average complexity, the advantage that bubble sort has over most other implementations, even *QuickSort*, is that it is able to detect that the list is already sorted. However, for our purpose this advantage is not relevant at all, but nevertheless this algorithm provides a very suitable structure for our exemplifications. The traditional method of this algorithm is shown in Listing 3.10.

Listing 3.10: Traditional Version of Bubble Sort

```
1  static void bubble1(int a[])
2  {
3    int i, j, t;
4    for(i=SIZE-1; i>0; i--)
5    {
6      for(j=1; j<=i; j++)
7      {
8        if (a[j-1] > a[j])
9        {
10         t = a[j];
11         a[j] = a[j-1];
12         a[j-1] = t;
13       }
14     }
15   }
16 }
```

Depending on the results of the branching condition, the iteration of the inner loop takes different times, which results in an execution-time variability. Hence, the control flow of this traditional algorithm strongly depends on the input-data.

In order to show how this problem can be overcome, we want to demonstrate the WCET-oriented variant of the *Bubble Sort* algorithm, introduced in [3] where we can see the changes made, even though the basic idea of the algorithm did not change. While the traditional algorithm still suffers from the execution-time variability due to the branching statement within the inner loop body, the modified algorithm avoids this problem by using two non-jumping conditional assignment operations. This algorithm is shown in the Listing 3.11.

Listing 3.11: Single-path Version of Bubble Sort

```
1  static void bubble2(int a[])
2  {
3    int i, j, s, t;
4    for(i=SIZE-1; i>0; i--)
5    {
6      for(j=1; j<=i; j++)
7      {
8        s = a[j-1];
9        t = a[j];
10       a[j-1] = ((s<=t) # s : t);
11       a[j] = ((s>t) # s : t);
12     }
13   }
14 }
```

The sole difference between those two algorithms is basically the way of swapping between the elements contained in the array. The single-path version is doing that in a way that elements being compared either get swapped or just overwritten by themselves. This is achieved by using a second variable in order

to also store the second value which is compared. Thus, it is possible to simply overwrite both values when indicated by the two conditions in line 10 and 11 of the given listing above.

Summing up, the if-statement of the traditional variant has been replaced by two constant-time conditional expressions which are implemented by four conditional move instructions and thus the implementation is totally free of conditional branches in the control flow.

3.3.4.4 Binary Search Algorithm

The next suitable algorithm which has quite a lot of potential to be adapted according to our requirmentes is the well-known *Binary Search* algorithm. This is an algorithm for locating the position of an element in a sorted list which finds the target value in logarithmic time. It uses a method to reduce the number of elements needed to be explored every round by a factor of two and therefore has both worst-case as well as average complexity $O(log\ n)$.

The first algorithm we are going to present here is one of the most basic and traditional variants, which is quite similar to iterative variants used in various standard libraries. This algorithm is shown in the following Listing 3.12.

Listing 3.12: Traditional Version of Binary Sort

```
1  static int bs_trad(int*a, int size, int key)
2  {
3    int left = 0;
4    int right = size - 1;
5    int idx;
6    do
7    {
8      idx = (right + left) >> 1;
9      if (a[idx] == key)
10       return idx;
11     else if (a[idx] < key)
12       left = idx+1;
13     else
14       right = idx-1;
15   }
16   while (right >= left);
17   return -1;
18 }
```

The strategy of binary search is quite intuitive and performs as follows. It starts by checking the middle, and continuously eliminates the half of the list from consideration in order to perform the search only on the remaining half. There are lots of different implementations of *Binary Search* which differ regarding their return value and are either recursive or iterative variants and also may vary in

their structure concerning the number of comparisons per iteration. They also differ regarding the fact whether the result is returned before or after the whole data has been explored.

If we examine the given variant, we notice that this algorithm suffers from several input dependecies. Due to the termination condition of the loop the algorithm has a variable iteration count. Furthermore, nested if-then-else constructs are contained in the algorithm and influence the control flow of the program-code in different ways.

In contrast, the WCET-oriented variant without any input dependencies, introduced in [4], is shown in Listing 3.13. In order to overcome the issues contained in the traditional variant, we note at this point that the main steps, introduced in section 3.3.4.2 , have again been performed in order to achieve an adapted variant of the binary sort algorithm.

Listing 3.13: WCET-oriented Version of Binary Sort

```
1  static int bs_singlep(int*keys, int size, int key)
2  {
3    int left = 0;
4    int right = size-1;
5    int idx = (right + left) >> 1;
6    for(int inc = size; inc > 0; inc >>= 1)
7    {
8      right = (key < keys[idx] # idx-1 : right);
9      left  = (key > keys[idx] # idx+1 : left);
10     idx   = (right + left) >> 1;
11   }
12   return (keys[idx] == key) ? idx : -1;
13 }
```

In order to be able to reproduce the differences between these two variants, we want to highlight some details of the adaption given in Listing 3.13:

- If we compare the two algorithms of Listing 3.12 and Listing 3.13 we can see the modification regarding the direct comparison to the searched key. The WCET-oriented variant processes the whole scenario and determines only at the end whether the key is contained in the sorted list. As a consequence, the nested if-condition can be dissolved.

- Since the WCET-oriented variant is now supposed to process the whole array, due to the changes made on the if-conditions, the number of iterations of the loop can be fixed to the maximum.

- Finally, applying the paradigm of if-conversion which is denoted by the already introduced *"#"* operator, helps us to overcome the last problems of data

dependencies. Again, this is achieved by overwriting the variable, which is either *right* or *left*, depending on which side was chosen to be searched next.

- The basic idea of the adapted algorithm is based on the fact that considering Line 8-9 in the Listing, the values of the variables *right* and *left* remain the same if the searched key has already been found before. As a consequence, the value of variable *idx* in Line 10 will remain the same up to the point at which the program ends. In order to implement the return value for not finding the element *"-1"*, the principle of a constant-time conditional expression is applied again and by doing so we achieve a single-path code showing the same functionality as the traditional variant.

3.3.4.5 Multi-Byte Counter

The last algorithm of that algorithm-class we want to present is the so-called *Multi-Byte Counter*. This algorithm increments a counter that consists of an array of single bytes. The counter increments by starting from the least significant element and iterates as long as there is an overflow to higher-order parts of the counter. Both algorithms which we are going to present in the following, are introduced in [6].

The functionality of this algorithm is quite intuitve and is basically implemented by one for-loop which iterates over all bytes of the given counter array as long as an overflow occurs at the current array position. The first time there is no overflow the loop terminates, which obviously violates our attempts for a single execution path. The behavior of this algorithm can be examined in the following Listing 3.14.

Listing 3.14: Traditional Version of Multi-Byte Counter

```
1  static void inc_counter_trad(COUNTER counter)
2  {
3    int idx;
4    for(idx=0; idx<COUNTERSIZE; idx++)
5    {
6     unsigned char tmp;
7     tmp = counter[idx];
8     counter[idx] = tmp+1;
9     if (counter[idx] > 0)
10       break;
11   }
12 }
```

A reasonable alternative to the previous algorithm that fulfills our needs for single-path code, is illustrated in the following Listing 3.15. The main difference

here lies in the fact that the value used for incrementing is defined by a variable. The variable gets set to zero as soon as no overflow occurs. Since the value zero of the variable cannot cause an effect anymore, the loop can pass through completely.

Listing 3.15: Single Path Version of Multi-Byte Counter

```
1  static void inc_counter_sp(COUNTER counter)
2  {
3    int idx, inc_val;
4    inc_val = 1;
5    for(idx=0; idx<COUNTERSIZE; idx++)
6    {
7      unsigned char tmp;
8      tmp = counter[idx];
9      counter[idx] = tmp + inc_val;
10     if (counter[idx] > 0)
11        inc_val = 0;
12   }
13 }
```

3.3.5 WCET Analysis with the Single Path Approach

The impacts for WCET analysis when using a single-path approach are very beneficial regarding the costs and effort. If we consider usual approches of WCET analysis, we recognize the following: First of all, path analysis becomes needless since the execution path of any execution with any given input data will always result in the same unique execution path. Furthermore, usage of complex hardware timing models is not necessary anymore, a fact which makes WCET analysis much easier, especially because of the fact that building such timing models requires a huge effort. The main reason for performing static analysis using such complex timing models is based upon the fact that a measurement-based analysis was not feasible, since the number of possible execution paths was simply too high. But when having only one execution path, measurements become possible again and then the WCET can be obtained by simply measuring the time of the single execution path.

3.4 Classification of Algorithms

The algorithms, presented in Section 3.4.1.1 are obviously part of a special category regarding their complexity and character. More precisely, these kind of algorithms are in the class of sorting and search algorithms of the superclass

of combinatorial algorithms. Almost every algorithm of the class of sorting algorithms is a comparison sort algorithm, since only comparisons are used to operate on the elements. The same holds for several algorithms of the class of search algorithms.

Applying the single-path approach to this kind of comparison-based algorithms is actually quite unproblematic, under the condition that an appropriate single-path solution for the given algorithm can be found.

But in contrast to that, there are many other classes of algorithms contained in different superclasses, comprising several operations which make the implementation of the exemplified single-path approach partly impossible. This affects basically mathematical operations (that is basic arithmetics), such as multiplication, division or mod-division and, moreover, some more sophisticated operations, based on these more basic operations.

In the following section and especially in Section 3.5.3 we illustrate some alternative ways for some numeric problems, respectively algorithms in order to maintain the property of uniform execution times and single-path code.

3.4.1 Achieving Single-Path Code by applying LookUp Tables

Intuitively we would say that we could bypass the problem of varying computation times if we preprocess all possible calculations and store the results somewhere, in order to be able to retrieve them in constant time. Since it is generally not possible to compute and store the results for the complete and possibly infinite set of input data, such an approach is only suitable if the domain of the input set can be limited to a specific bound. Therefore, we want to introduce the well-known technique of **LookUp Tables** which allows us to store the data needed in order to process a given request for a particular problem respectively algorithm.

Before the arising of computers, printed lookup tables of values were used by people to speed up hand calculations of complex functions, such as in trigonometry, logarithms, and statistical density functions. In computer science, a lookup table is a data structure, usually an array that is often used to replace runtime computations with a simpler array indexing operation. This technique is typically used for achieving significant savings of processing time. For our concerns we are primarily focussed on uniform execution times respectively on single-path code.

3.4.1.1 Population Count

A famous example for the usage of a look-up table is the problem of *Population Count*, also known as *Hamming Weight*. It is a discrete problem, counting the number of bits which are set to '1' in a (binary) number. The following Listing 3.16 shows the basic example of C code, designed to count the number of bits set to '1'.

Listing 3.16: Counting '1' bits in a series of bytes

```
1 int count_ones(unsigned int x)
2 {
3     int result = 0;
4     while (x != 0)
5         result++, x = x & (x-1);
6     return result;
7 }
```

The presented algorithm can take a varying number of cycles depending on the input given to that algorithm. However, there exists a faster but uniform solution, using a trivial hash function table lookup, which is shown in the following Listing 3.17

Listing 3.17: Counting '1' bits with Look-up Table

```
1 /* Code for 'int' 32-bits wide */
2  int count_ones(unsigned int x)
3 {
4     return bits_set[ x         & 255] +
5            bits_set[(x >>  8) & 255] +
6            bits_set[(x >> 16) & 255] +
7            bits_set[(x >> 24) & 255];
8 }
```

Here the static table *bits_set* with 256 entries is constructed. The table provides for a certain input-value the number of bits which are set to one in order to represent that value. Then this table can be used to find the number of ones in each byte of the integer, using a simple look-up function for each of its bytes and finally accumulate the sum of them.

3.4.2 Single-Path Code with Arithmetic Influences

If we consider the problems mentioned in Section 3.4, we will notice that there must be plenty of algorithms which are suitable for adapting them to single-path code, but on the other hand have some additional functions which are not solvable with the methods and strategies presented in Section 3.3.3.

Therefore, we are going to discuss the problem of the $Greatest\ Common\ Divisor$, which is a good example of an algorithm which does not fit too well for single-path code. We want to address two different algorithms, which are first of all the standard implementation of Euclid's Algorithm and, secondly, the binary GCD algorithm, also known as Stein's algorithm.

3.4.2.1 Standard (Euclid's) GCD Algorithm

The Euclidean algorithm, given in Listing 3.18, is one of the most efficient and most used algorithms for computing the greatest common divisor. Nevertheless, the algorithm is not well suited for our single-path programming style. This is because of the fact that the number of loop iterations depends on the size of the input numbers. If we fix the number of loop iterations to a certain high bound in order to cover a huge range of input data, the problem of the modulo-division remains. The problem in particular concerns the fact that hardware often does not support the division operation whereas the available software-division has variable execution times.

Listing 3.18: Standard implementation of GCD

```
1  int gcd(int a, int b)
2  {
3     if(a < 0) a = -a;
4     if(b < 0) b = -b;
5     while( b > 0)
6     {
7       int temp = a;
8       a = b;
9       b = temp % b;
10    }
11    return a;
12 }
```

In order to discuss the number of maximal loop iterations we want to consider the worst case situation which occurs when the input paramters a and b are two adjacent Fibonacci numbers, because in that case the remainders follow the Fibonacci sequence until zero is obtained. This is due to the relation between the nature of Fibonacci numbers and the modulo-division.

We want to consider adjacent Fibonacci numbers (starting with F_0 = 0) that we use for the single-path algorithm, given in the Listing 3.19. The given numbers are within a range, until the upper bound for signed integers ($2^{31} - 1$). This would include numbers up to (and including) the 46th Fibonacci number.

Listing 3.19: Single-path variant of GCD

```
1  int gcd_sp(int a, int b, int n)
2  {
3    a = a < 0 ? -a : a;
4    b = b < 0 ? -b : b;
5    int temp;
6    for(int i = 0; i < n; i++)
7     {
8       temp = (b > 0) ? a : 0;
9       a = (b > 0) ? b : a;
10      b = temp % b; /* optionally Soft-Mod-Div. */
11    }
12   return a;
13 }
```

In our experiments, treated in Section 4, we will point out in all details the relation between the input data and the required number of loop iterations.

3.4.2.2 Binary GCD Algorithm

In contrast to the ancient Euclidean algorithm the use of the binary GCD algorithm has the advantage that the (modulo)division can be replaced by shifts. This is quite beneficial for embedded platforms which often do not have direct processor support for division. The algorithm was first published by the Israeli physicist and programmer Josef Stein in 1967, but has been probably already known in first-century in China.

The algorithm, given in Listing 3.20, solves the problem of finding the GCD by repeatedly applying the following rules:

1. Since everything divides zero and v is the largest number that divides v, $gcd(0, v) = v$ holds. Similarly for $gcd(u, 0) = u$. Although $gcd(0, 0)$ is not defined, it is convenient to define $gcd(0, 0) = 0$.

2. If u and v are both even, then $gcd(u, v) = 2gcd(u/2, v/2)$ holds since 2 is a common divisor. Additionally, this step yields to the variable k which is the number of common factors of 2.

3. If u is even and v is odd, then $gcd(u, v) = gcd(u/2, v)$ holds since 2 is not a common divisor. For inverted parity of u and v naturally $gcd(u, v) = gcd(u, v/2)$ holds.

4. If u and v are both odd, and $v \geq u$, then $gcd(v, u) = gcd((v - u)/2, u)$ holds. If both are odd and $v < u$, then $gcd(v, u) = gcd((u - v)/2, v)$ holds. Obviously, a division by 2 can only result in an integer since the difference of two odd numbers must be always even.

5. By repeating steps 3-4 until $v = u$ or $v = 0$ the algorithm yields the result which is $2^k * u$ (where variable k is determined by the second step)

Listing 3.20: Binaray GCD Algorithm

```
/* Function to determine number to be even or odd */
#define EVEN(x) ((x&0x1)==0)

int gcd_binary(int a, int b)
{
  unsigned int u = (a < 0) ? (-a) : a;
  unsigned int v = (b < 0) ? (-b) : b;
  int k = 0;
  /*  GCD (0,x) := x  */
  if(u == 0 || v == 0) return (u | v);

  while(EVEN(u)){
    u = u >> 1;
    if(EVEN(v)){
      v = v >> 1;
      k++;
    }
  }
  /* From here on, u is always odd. */
  while (v > 0) {
    while(EVEN(v)) v = v >> 1;
    if(u < v) v = v - u;
    else {
      int diff = u - v;
      u = v;
      v = diff;
    }
  }
  return u << k; /* u*(2^k) */
}
```

The following algorithm, given in Listing 3.21, shows the transformated single-path variant of the binary GCD algorithm. Here the number of loop iterations is bounded by values which restrict the range of the inputs to 32-bit signed integers.

Listing 3.21: Binary Single-path variant

```
int gcd_binary_sp(int a, int b)
{
  unsigned int u = (a < 0) ? (-a) : a;
  unsigned int v = (b < 0) ? (-b) : b;
  int k = 0;
  /*  GCD (0,x) := x  */
  if(u == 0 || v == 0) { u = v = (u | v);}
  int i;

  for ( i = 0; i < 31; i++){
     int even_u = (u & 0x1) == 0;
     int even_v = (v & 0x1) == 0;
```

```
13      if (even_u) u = u >> 1;
14      if (even_v) v = v >> 1;
15      if (even_u && even_v) k++;
16    }
17    for (i = 0; i < 60; ++i) {
18      int diff = u - v;
19      if ((v & 0x1) && diff < 0) v = -diff;
20      if ((v & 0x1) && diff >= 0) u = v;
21      if ((v & 0x1) && diff >= 0) v = diff;
22      v = v >> 1;
23    }
24    return u << k; /* u*(2^k) */
25  }
```

3.5 Trigonometric Functions

In mathematics, the trigonometric functions are functions of an angle and used to relate the angles of the triangle to the lengths of its sides. Trigonometric functions are essential when modeling periodic phenomena and are also present among many other applications. The most familiar ones are the *sinus*, *cosine* and *tangent* functions.

For now, we want to concentrate on the *cosine* since it is an symmetric function and which is mostly used to derive the results of all other trigonometric functions. Later on, we are going to show a proper calculation for the *consine* provided by an example using look-up tables in order to bypass the problem of varying execution times.

3.5.1 Iterative Approximation using Tayler Series

At first, we want to show the common implementation where Taylor Series are used to approximate the trigonometric functions. There are also other iterative algorithms such as CORDIC [17] with no need of multiplication operations, which are thus more suitable for their implementation in hardware.

Since we are discussing software implementations, we will briefly illustrate the Taylor Series. The following formula 3.5.1 shows the definition of the Taylor Series for the *cosine* approximation. In order to reach full *double precision* (floating point format), a Taylor polynom of degree 14 would be required [18]. Considering the given formula, there remain only the 7 coefficients of even powers, which need to be calculated.

$$cos(x) \approx \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!} = \frac{x^0}{(0)!} - \frac{x^2}{(2)!} + \frac{x^4}{(4)!} - .. \tag{3.5.1}$$

An example for an approximate calculation of the *cosine* with an accurateness up to 6 decimals is shown in the following Listing 3.22. It illustrates how the theory of Taylor Series can be used for calculating the *cosine* for real numbers of x and it should also exemplify the basic mathematical concept and their complexity of Taylor series, used in standard C-libraries such as *math.c*.

Obviously, implementations like this one, using arithmetic operations in order to calculate the result, suffer from varying execution times, since the complexity of processing some arbitrary input is not constant.

Listing 3.22: Taylor Series - cosine approximation

```
1  float cos(float a)
2  {
3   /* Taylor series for cosinus calculation:
4      cos(x) = (x^0/0!)-(x^2/2!)+(x^4/4!)-(x^6/6!)+...   */
5
6      float i = 0.0;
7      i = (pow(a,0)/1) -
8          (pow(a,2)/2) +
9          (pow(a,4)/24) -
10         (pow(a,6)/720) +
11         (pow(a,8)/40320) -
12         (pow(a,10)/3628800) +
13         (pow(a,12)/479001600);
14     return i;
15 }
```

3.5.2 Approximation using Lookup Tables

In order to get rid of the problem of varying execution times, we propose the well-known approach of look-up tables, introduced in Section 3.4.1, in combination with interpolation. We could abandon the interpolation if the amount of data-storage was boundless but we will rather have to cope with limited space for data of the lookup table which makes the use of interpolation inevitable. Although there are different types of interpolation, we apply a basic linear interpolation which fulfils our purpose of demonstrating an alternative approach.

The following code in Listing 3.23 shows the pure code containing the mathematical approach for our calculations which we are finally going to use for our experiments. The approach comprises fixed-point arithmetic in order to use the

data type of *Integer* and furthermore includes several functions for multiplication, division and modulo-division which are written in a single-path manner. The following variant is designed only for a range of values between 0 and $\frac{\pi}{2}$. In our experiments, treated in Section 4, we will show some other adaptions of that algorithm which support an extended range of input-values.

Listing 3.23: Algorithm using LookupTable for cosine calculation

```
1  int cos_(int a)
2  {
3    volatile int y, y1, y2;
4    volatile int x, x1, x2, m_, m__, comma;
5
6    /* divide through 1024 */
7    x =  (a >> 10);
8    /* modulo function with dividend a and divisor 1024  */
9    m_ = mod_1024_SP(a);
10   /* multiplication function */
11   m__ = MUL_(10000, m_);
12   /* divide through 1024 for getting 4-digit comma */
13   comma = (m__ >> 10);
14
15   /* lookup-values for array (index) */
16   x1 = x;
17   x2 = x+1;
18   y1 = LUp_table[x1];
19   y2 = LUp_table[x2];
20
21   /* Interpolation function containing MUL_ and DIV_ */
22   y = ppoint(y1,y2,comma);
23
24   return y;
25 }
```

The algorithm starts basically with evaluating the first index-value of the lookup table, which is x_1, by performing an integer division. That means that the input value a - a fixed point number - is divided by 1024 (line 7). Naturally, we know that the real value which needs to be calculated has to be somewhere between the retrieved values of $LUp\_table[x_1]$ and $LUp\_table[x_2]$ (lines 18-19). In order to determine the decimal places ($comma$) of the division, we need to perform a modulo-division with a subsequent multiplication and division (lines 9, 11 and 13). The value can finally be used to call the interpolation function (line 22) which returns the approximate value for the desired input value.

The function shown in Listing 3.24, calculates the real value y, which lies between the first two parameters y_1 and y_2. At the beginning, the function evaluates if the desired value resides within an increasing or decreasing section of the curve, which is recorded by the variable $course$. The third parameter x is used to determine the relative distance between y and y_1, denoted by $y_{\_dist}$

(line 12 and 13). The latter is finally either added to y_1 or substracted from it, depending on the value of the variable $course$ (line 15 - 17).

Listing 3.24: Linear Interpolation Function

```
1  int ppoint(int y1, int y2, int x)
2  {
3      volatile int course, y, e1, e2;
4      volatile uint32t y_, y_dist;
5      volatile int dist;
6
7      course = y1 - y2;
8      e1 = y1 - y2;
9      e2 = y2 - y1;
10     dist = (course > 0) ? e1 : e2;
11
12     y_ = MUL_(dist,x);
13     y_dist = udiv_SP(y_, 10000);
14
15     e1 = y1 - y_dist;
16     e2 = y1 + y_dist;
17     y = (course > 0) ? e1 : e2;
18
19     return y;
20 }
```

Even though the functions described in the Listings 3.23 and 3.24 are written in a single-path manner, there remain several problems in the code that need to be solved in order to achieve uniform computation times for given input values. This includes the matters of arithmetic operations contained in both functions, but also the issue of avoiding soft float libraries supporting the floating point arithmetic, which is often used for such trigonometric calculations. Several arithmetic algorithms and an alternative solution for floating point arithmetic are presented in more detail in Section 3.5.3.

The code presented in Listing 3.23 uses an array containing the stored values of the lookup table. The following pseudo code (Listing 3.25) shows how to construct the table and furthermore gives information about the exact structure and model chosen for the table.

Listing 3.25: Generation of LookUp-Table for cosine

```
1 /* filling table with cos-values for interval 0-PI/2 */
2 double PR = 0.0559529096807444
3 double DTR = (PI * PR) / 180
4 double RTD = 1/DTR = 1024.00000000
5 double cos_value = 0.0
6 int i
7 int cosLUT[1609] /*array size of 1609*/
```

```
8
9  FOR LOOP with i = 0..1609
10 {   /* calculate cos values in steps of 1/17.872175 degree */
11     cos_value = (double)cos(i * DTR);
12     /* converte to Q7.24 format and store */
13     cosLUT[i] = (int)( round(cos_value*(2^24)) );
14 }
```

The value 1024 of the variable 'RTD' which is the reciprocal value of the used variable 'DTR' in line 11 demonstrates our lookup strategy that we use in our algorithm in Listing 3.23. Considering this algorithm, we can see that the lookup table is generated in a way that we are able to resolve our values contained in the table (array) by simply dividing our input value through 1024. Since 1024 is a power of two the division operation is realized by a ten bit shift to the right.

3.5.3 Strategies and Functions

This section describes some techniques of avoiding soft-float libraries and furthermore provides some numeric algorithms which are needed in order to substitute the required arithmetic operations of the illustrated cosine function described in the previous Section 3.5.2. Since the class of numeric algorithms contains the classical arithmetic operations such as multiplication, division or modulo-division, we are forced to deal with the variations of computing times. We are going to show that it is possible to implement arithmetic functions which are partly adapted to the properties of the given target hardware and which are able to overcome the issues of variable execution times for different operations.

3.5.3.1 Fixed-point Arithmethic instead of Floating Point

Floating point units (FPU) are integrated into all the major microprocessors used in standard PCs, so that one would assume fast floating point number calculations to be granted. But even now many DSPs (Digital Signal Processors) used in embedded systems do not have an FPU. Since most low-end embedded systems still do not have a hardware FPU for performing floating point arithmetic, C-compilers usually provide floating point support by soft-float libraries.

One major concern regarding single-path code and uniform computation times is at any rate to get rid of useless complexity of source code respectively low-level machine code. The use of such libraries makes the whole process not only significantly slower but also yields an increasing complexity of low-level code which might violate our needs of single-path code and uniform computation times. Due

to the first reason, the reason of speed, embedded projects often require a 'no floating point rule' on their programmers, while the second issue concerns our fundamental idea of predictable programming code.

Hence, in order to overcome this problem and generate program code that satisfies our needs, we are forced to deal with fix-pointed arithmetics. We want to show now how to convert a floating point number into a fixed-point number. The formula for calculating the integer representation X of a float number x is given in formula 3.5.2.

$$X = round(x * 2^n) \tag{3.5.2}$$

The formula for converting the integer representation back to a float number is given in formula 3.5.3.

$$x = X * 2^{-n} \tag{3.5.3}$$

Depending on the number of available bits, there can be set different formats for fixed-point numbers. If we assume to have 16 bit available, an common option for a partitioning into integer places and decimal places would be the following:

- Considering our equations above - if we assumed n = 8 - this would imply an assignment of 8 bit to the fractional part.
- Of the remaining 8 bit we can take at most 7 bit for representing the integer part.
- The last bit remains for the sign.
- Such an partitioning would be denoted as Q7.8 format.

For our table-lookup algorithm presented in Listing 3.23 we chose two different formats. Having 32 bit available, we chose the Q7.24 format for the values stored in the lookup table to be Q7.24. The format for the input data (values to be calculated) was chosen to be Q11.20. With the chosen formats we obtained the desired accurray for the representation of the floating point decimal places which was fixed to 7 places for the lookup-table values and to 4 places for the value of the input data.

3.5.3.2 Single-path Multiplication Algorithm

The task of our algorithm is to execution times for multiplications for integer numbers within a range of 32 bit. This means that the algorithm is able to

take two arbitrary input values with a magnitude of 1 to 4 bytes (up to 32 bit), multiplies them and returns the result within a constant computation time that is a constant number of CPU cylces.

The multiplication algorithm is constructed according to the target hardware used for our experiments. The construction is based on the fact that the underlying processor operates with a 32 bit x 8 bit multiplier. The standard implementation of the multiplication yields different execution times, depending on the number of bytes used for the input values. Let us assume, for instance, two values x and y and further assume that x is assigned to the larger 32 bit register of the multiplier holding an arbitrary value of up to 32 bit. Then we can expect to get an overhead of execution cycles, depending on the size of the second value y which is assigned to the remaining smaller register.

The following implementation, given in Listing 3.26, overcomes this problem by multiplying each single byte of the first value with each other byte of the second value. Shifting the partial results to the correct position leads to the final result of the entire multiplication.

Listing 3.26: Single-Path Multiplication Algorithm

```
1  int MUL_(int x, int y)
2  {
3      if(y < 0) { y=-y;x=-x; }
4      int y0 = y & 0xFF;          /* 1. byte of y */
5      int y1 = (y>>8) & 0xFF;     /* 2. byte of y */
6      int y2 = (y>>16) & 0xFF;    /* 3. byte of y */
7      int y3 = (y>>24) & 0xFF;    /* 4. byte of y */
8      int z = y0*x;
9
10     z += (y1*x) << 8;
11     z += (y2*x) << 16;
12     z += (y3*x) << 24;
13
14     return z;
15 }
```

3.5.3.3 Single-path Integer Division Algorithm

The main requirement of a single-path integer division is to achieve uniform execution cycles for the division of two integer numbers within a range of 32 bit. The algorithm we are going to present here originates from $Ian\ Kaplan$ and has been adapted by $Benedikt\ Huber$ in order to fulfill our request of constant execution cycles. The original algorithm can be examined in Section A.2. The main reference, used by $IanKaplan$, for implementing the algorithm was the book $Digital\ ComputerArithmetic$ by $J.F.Cavanagh$ [19].

The presented algorithm is a so-called $radix\ two$ division algorithm where one computation step is needed for each binary digit. Besides, there are also radix 4, 8, 16 and even 256 algorithms, which are faster, but at the same time more difficult to implement.

Listing 3.27: Single-Path Division Algorithm

```
#define LAST_BIT_INDEX 31
#define LAST_BIT_MASK  0x80000000
#define FULL_BIT_MASK  0xFFFFFFFF

uint32_t udiv_SP(uint32_t dividend, uint32_t divisor)
{
    uint32_t t, flag = 0;
    uint32_t q, bit;
    int i;
    uint32_t remainder = 0;
    uint32_t quotient = 0;

    for (i = 0; i < (LAST_BIT_INDEX+1); i++) {
        bit = (dividend & LAST_BIT_MASK) >> LAST_BIT_INDEX;
        remainder = (remainder << 1) | bit;
        dividend = dividend << 1;
        t = remainder - divisor;
        q = !((t & LAST_BIT_MASK) >> LAST_BIT_INDEX);
        if(remainder >= divisor) flag = FULL_BIT_MASK;
        quotient = (quotient << 1) | (q & flag);
        if (q & flag) remainder = t;
    }
    return quotient;
}
```

3.5.3.4 Single-path Integer Modulo Division Algorithm

Related to the integer division is the integer modulo division which comprises the two algorithms introduced in Section 3.5.3.2 and Section 3.5.3.3. The implementation is straightforward, which is why we do not go into more detail.

Listing 3.28: Single-Path Modulo Division Algorithm

```
uint32_t mod_SP(uint32_t dividend, uint32_t divisor)
{
    volatile uint32_t q,qq,d = 0;
    q = udiv_SP(dividend, divisor);
    qq = MUL_(divisor,q);

    return dividend - qq;
}
```

Chapter 4

Experiments

4.1 Overview

In this chapter we present the results of our experiments and we show how the programming strategies and programming paradigms described in the sections before can be applied on a selected embedded hardware. The experiments contain several comparisons of the selected algorithms that have been described in Section 3.3.

4.2 Basic Conditions

The test environment for our experiments comprises not only a particular manner of programming style, but also a specific hardware in combination with compiler. Furthermore, we discuss the form and the dimension of the measured execution time when evaluating several algorithms and also refer to additional tools which are used within our experiments.

4.2.1 Test - Environment

The environment used for our experiments consists of an ARM7-TDMI on an ST730 evaluation board and the IAR Embedded Workbench IDE, which is a powerful Integrated Development Environment, allowing to develop embedded application projects.

The ARM7-TDMI processor is a 32-bit RISC CPU designed by ARM, which is a versatile processor designed for mobile devices and other low power electronics. The processor supports both 32-bit and 16-bit instructions via the ARM and Thumb instruction sets. The latter is a subset of the ARM instruction set and

permits higher code density (smaller memory requirement).

ARM7-TDMI instructions can be executed conditionally and therefore have a 4-bit condition field in the instruction. The state of the condition flags can be checked and if the condition flag state matches the condition, the instruction executes normally. If the condition flag state does not match the condition, the instruction is executed as a NOP (no operation).

For the realization of the single-path programming paradigm we require a processor that provides a conditional move instruction with constant execution time and the ARM7-TDMI processor satisfies this requirement.

The Development Environment, the IAR Embedded Workbench IDE framework, incorporates the following tools:

- ARM IAR C/C++ Compiler
- ARM IAR Assembler
- versatile IAR ILINK Linker
- Editor and Project Manager
- IAR C-SPY debugger - state-of-the-art high-level language debugger

The IAR Embedded Workbench IDE comes with functions that allow programmer to stay in control of all project modules, for example, C or C++ source code files, assembler files, include files, and other related modules.

The ARM IAR C/C++ Compiler is a state-of-the-art compiler that offers the standard features of the C or C++ languages, plus some extensions designed in order to take advantage of the ARM-specific facilities.

4.2.2 Measuring Execution Time

For our experiments the execution time for a given program p is defined by the number of $CPU\ clock\ cycles$ required by that program. Since we are not forced to present results in terms of physical time but rather need to compare different implementations with respect to their variability and offset, we are satisfied with the term of $CPU\ clock\ cycles$.

For determining the exact number of clock cycles, an internal timer was used for performing the measurements. We can express our indiviual notion of measured execution time for a program p if we just simplify the first formula of Section 2.3.2:

$$Execution\ time_{Cycles} = CPU\ clock\ cycles_{Programm\ p} \quad (4.2.1)$$

The number of clock cylces was determinded by connecting the RS232 interface of the ST730 evaluation board to a PC. The data communication was established by using the Microsoft Hyper Terminal (Version 5.1) which used the adjustments of our test framework that was generated with the development environment. The RS232 interface was configured on both sides and the settings have been adjusted to a baudrate of 9600 with 8 data bits, even parity and one stop bit.

4.2.3 Generation of Test Data

In order to perform the experiments with our algorithms, we have to generate appropriate random test data. We used the $rand()$ function of MATLAB R2007, which produces uniformly distributed pseudorandom numbers. MATLAB R2007 is a numerical computing environment, devloped by $TheMathWorks$, and is highly used across industry and the academic world.

4.3 Experimental Code and Results

In this section we present our results and measurements obtained by our experiments. Regarding the straight comparision-based class of algorithms (including FindFirst, Bubble Sort, Binary Search and the Multi-Byte Counter), we illustrate the source code of the FindFirst algorithm. Examining the code, we want to point out the difference between the real syntax which we used in the source code of the experiments and the one presented in Section 3.4.1.1. The essential difference lies exclusively in the used conditional operator (using "?" in the real code instead of "#"), whereas the other parts of code of these algorithms are identical with the source code of our experiments.

We also show the source code for the variants of the GCD algorithm and furthermore, we are going to present additional variants for the $cosine$ algorithm that support an extended range of input-values.

The assembler code for all algorithms can be examined in the Section 5.

4.3.1 FindFirst

4.3.1.1 Variants of Find First Algorithms

Listing 4.1: Find First - Traditional variant

```
1  static int findfirst1(int *keys, int size, int key)
2  {
3    int i;
4    int position = -1;
5    for(i=0; i<=size-1; i++)
6    {
7      if (keys[i] == key) /* max. iterations: SIZE */
8      {
9       position = i;
10      break;
11     }
12   }
13   return position;
14 }
```

Listing 4.2: Find First - Single-path variant

```
1  static int findfirst2(int *keys, int size, int key)
2  {
3    int i;
4    int position = -1;
5    for(i=size-1; i>=0; i--) /* max. iterations: SIZE */
6      {
7       position = ((keys[i] == key) ? i : position);
8      }
9    return position;
10 }
```

4.3.1.2 Results and Measurements

For the experiments with the FindFirst algorithm we provided a list of 128 randomly chosen values within the range between 0 and 100. Since the array has just a size of 128, we got single values to be in the list more than once. Based on this list, we performed a "find first" search with 102 iteration runs including all values between 0 and 100. On average, each third element was not contained in the list. The execution time in Version 1, for the scenario of not finding the searched element in the list, yields the WCET. The different execution times can be examined in Table 4.4.

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
1	Traditional Variant	858.27	154	1430	1
2	Single Path Variant	1551	1551	1551	1.0846

Table 4.1: Results of FindFirst Experiments

Considering now our results in Table 4.4, we see that unlike Version 1, the other implementation, Version 2, shows a constant execution time, no matter

the element is contained in the list or not. We have to note that the constancy is only given if the size of the list (array) is defined statically. The WCET factor of 1.0846 brings an overhead of execution time which seems to be insignificant. Thus, the single-path variant of the FindFirst algorithm can be seen as a quite suitable alternative to the traditional variant.

4.3.2 Bubble Sort

4.3.2.1 Results and Measurements

For the experiments with the Bubble Sort algorithm we chose our test data to be an array of the size 16. For this array, we chose a sequence of 16 distinct values. In order to achieve 50 different sequences of numbers, we evaluated 50 different random permutations, including two special cases - the worst-case and best-case scenario. The two special cases were generated manually.

The worst case for the Bubble Sort algorithm is a list in descending order, whereas the best case is given by an already sorted (ascending) list. These cases could be confirmed in our experiments by the results for the WCET and BCET. Table 4.2 reveals the different execution times obtained by our measurments.

The source code of both algorithms, Version 1 and Version 2, can be looked up in Section 3.3.4.3.

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
1	Traditional Variant	2014.8	1895	2135	1
2	Single Path Variant	2495	2495	2495	1.1686

Table 4.2: Results of Bubble Sort Experiments

4.3.3 Binary Search

4.3.3.1 Results and Measurements

For the experiments with the Binary Search algorithm we provided an array of size 128 filled with data. Later on, we changed the size of the array for our measurments first from 128 to 64 and afterwards to 28.

In order to check all different cases during our testing, we used an array with 128 values including all even values between 0 and 254. With that array we performed our measurments with an input data set which covered all even and odd values within this range and some additional boundary values ouside

the range. For the experiments with an array-size of 64 respectivly 28, we downscaled the input data set proportionally.

Table 4.3 lists the different execution times obtained by our measurments. One can see that the WCETs of both Versions are very close to each other, which shows that the single-path variant (Version 2) is an adequate alternative compared to its original implementation (Version 1).

Similar to the Find First algorithm, also here the size of the list (array) needs to be defined statically in order to achieve constant computation times. Furtheremore, the results in the table show that it is not always the same algorithm that has the better WCET factor, which depends on the size of the array and not on the chosen input data.

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
Size 128					
1	Traditional Variant	247.37	151	274	1
2	Single Path Variant	269	269	269	0.9817
Size 64					
1	Traditional Variant	230.69	151	257	1
2	Single Path Variant	254	254	254	0.9883
Size 28					
1	Traditional Variant	212.39	151	223	1
2	Single Path Variant	224	224	224	1.0044

Table 4.3: Results of Binary Search Experiments

The source code of both algorithms, Version 1 and Version 2, can be found in Section 3.3.4.4.

4.3.4 Multi-Byte Counter

4.3.4.1 Results and Measurements

For the experiments with the Multi-Byte Counter we used an array of size 10, providing 10 bytes. According to the source code of the algorithm, presented in Section 3.3.4.5, we provided different array assingments, which covered all possible scenarios for the behavior of the algorithm. This included 10 different distinct assignments where the first one contained for every array position a value smaller than 255. The other ones were modifed in a way that from the one side to the other, always one additional array position was set to 255. This was done systematically until all values in the array except the last one were set to 255 (in order not to cause an overflow of the 10-byte counter). Furthermore,

we provided some additional random input values where at least one position of the array was set to 255.

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
1	Traditional Variant	232	160	304	1
2	Single Path Variant	276	276	276	0.9079

Table 4.4: Results of Multi-Byte Counter Experiments

4.3.5 Basic Arithmetic Functions

In the following Section, Section 4.3.5.1, we briefly show the execution times for the numeric single-path algorithms used within the cosine implementation. This concerns the integer multiplication and the integer division. The single-path integer modulo-division is based on the basic division and therefore does not need to be examined. Therefore, we provided only a small subset of random unsigned integer numbers within the range of 32 bit which should illustrate the different behaviors for the execution times of the respective variants.

4.3.5.1 Results and Measurements

Table 4.5 shows the execution times for the integer division and multiplication. One can see that the standard multiplication needs just a few cycles. This is because the underlying operation is implemented in hardware. The single-path variant has to be implemented in software which leads to a considerably larger execution time.

Both, the standard division and the single-path division algorithm are pure software implementations. One can see that the factor between these two variants is much smaller than the factor of the multiplications.

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
1	Standard MUL	5	4	7	1
2	Single Path MUL	156	156	156	26.5
1	Standard DIV	204.43	185	240	1
2	Single Path DIV	685	685	685	2.8541

Table 4.5: Results of Integer Multiplication and Division Experiments

The source code for both algorithms, Version 1 and Version 2, and a detailed explanation of the hardware properties that concern the implementation of the

single-path multiplication can be found in Section 3.5.3.2 respectively Section 3.5.3.3.

4.3.6 Greatest Common Divisor - Euclid

4.3.6.1 Variants of Standard GCD Algorithm

Listing 4.3: GCD - Standard Implementation

```
1  int gcd(int a, int b)
2  {
3      if(a < 0) a = -a;
4      if(b < 0) b = -b;
5      while( b > 0)
6      {
7        int temp = a;
8        a = b;
9        b = temp % b;
10     }
11     return a;
12 }
```

Listing 4.4: GCD - Single-path variant

```
1  int gcd_sp(int a, int b, int n)
2  {
3    a = a < 0 ? -a : a;
4    b = b < 0 ? -b : b;
5    int temp;
6    for(int i = 0; i < n; i++)
7     {
8       temp = (b > 0) ? a : 0;
9       a = (b > 0) ? b : a;
10      b = temp % b; /* optionally Soft-Mod-Div. */
11    }
12   return a;
13 }
```

For our experiments with the different GCD algorithms we chose a special subset of the Fibonacci numbers as our test data. Since the worst case for the Standard Euclid GCD algorithm is determined by two adjacent Fibonacci numbers, we included, among other numbers, the two biggest Fibonacci numbers which are within the range of signed integers ($2^{31} - 1$) in our test values. This included numbers till (and including) the 46th Fibonacci number. We denote the sequence number of the biggest Fibonacci number within the range of 32-bit signed integers to be N_{max} = 46.

Applying the two biggest adjacent Fibonacci numbers of that sequence to the algorithm shows that the loop variable n of the algorithm has to be chosen to be at least $n \geq N_{max} - 1$ = 45 in order to gain correct results for the greatest common divisor.

Furthermore we could observe that the variable n can be reduced by one if we ensure that the algorithm takes always the bigger number for the first argument

($a > b$). Otherwise, if ($a < b$), the algorithm needs exactly one extra loop-iteration in order to swap the two input parameters a and b.

4.3.6.2 Variants of Binary GCD Algorithm

Listing 4.5: Binary GCD Implementation

```
#define EVEN(x) ((x&0x1)==0)

int gcd_binary(int a, int b)
{
  unsigned int u = (a < 0) ? (-a) : a;
  unsigned int v = (b < 0) ? (-b) : b;
  int k = 0;
  if(u == 0 || v == 0) return (u | v);
  while(EVEN(u)){
    u = u >> 1;
    if(EVEN(v)){
      v = v >> 1;
      k++;
    }
  }
  while (v > 0) {
    while(EVEN(v)) v = v >> 1;
    if(u < v) v = v - u;
    else {
      int diff = u - v;
      u = v;
      v = diff;
    }
  }
  return u << k;
}
```

Listing 4.6: Binary GCD - Single-path variant

```
int gcd_binary_sp(int a, int b)
{
  unsigned int u = (a < 0) ? (-a) : a;
  unsigned int v = (b < 0) ? (-b) : b;
  int k = 0;
  if(u == 0 || v == 0) { u = v = (u | v);}
  int i;
  for ( i = 0; i < 31; i++){
     int even_u = (u & 0x1) == 0;
     int even_v = (v & 0x1) == 0;
     if (even_u) u = u >> 1;
     if (even_v) v = v >> 1;
     if (even_u && even_v) k++;
  }
  for (i = 0; i < 60; ++i) {
    int diff = u - v;
    if ((v & 0x1) && diff < 0) v = -diff;
    if ((v & 0x1) && diff >= 0) u = v;
    if ((v & 0x1) && diff >= 0) v = diff;
    v = v >> 1;
  }
  return u << k;
}
```

4.3.6.3 Results and Measurements

Table 4.6 shows the summary of our results of the GCD algorithms. Version 2a is a small modification of Version 2 which is shown in Listing 4.4. Here the basic modulo-division (Line 10) gets substituted by the $Software\ Modulo\ Divison$ that has been introduced in Section 3.5.3.4.

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
1	Standard	972	194	2240	1
2	Standard (Single Path Code)	1769	1380	2326	1.0384
2a	Standard (SP + SoftModulo)	27638	27638	27638	12.3384
3	Binary GCD	577	170	710	1
4	Binary GCD (Single Path Code)	1705	1705	1705	2.4042

Table 4.6: Results of different GCD implementations

The impacts on the execution time of the fixed loop bound when using the expensive software modulo-division can be seen clearly in the table above. Due to the long execution time of the modulo division, the whole algorithm appears to be quite inapplicable for further use. Only with an adequate solution for the modulo division the adaption of Euclid's algorithm makes sense. Although both the binary and the standard single-path algorithm have the problem of input-data dependent loop bounds, the binary variant (Version 4) brings practical results on the execution time which are also beneath the WCET of the standard Euclid algorithm (Version 1).

4.3.7 Trigonometric Functions - Cosine

4.3.7.1 Variants for the Cosine Implementations

The following two Listings, Listing 4.7 and 4.8, show how the original algorithm with the defined lookup table, illustrated in Listing 3.23, can be adapted in order to enable results for an extended range of input values that are not covered by the defined lookup table. The first Listing shows how to extend the interval of the original algorithm from [0, $\frac{\pi}{2}$] to the interval [-π, π]. If the code extensions are written in such an appropriate single-path oriented manner, the uniform execution time can be maintained.

Listing 4.7: Single Path Cosine - Interval: -PI to PI

```
1  /* cosine function: interval -PI to PI */
2  int cos_(int a)
3  {
4    volatile int y1, y2, y, y_minus;
5    volatile int x, x1, x2;
6    volatile int m_, m__, comma;
7    volatile int a_overPI_half, a_v;
8    volatile int a__ = a;
9    volatile int a_m = -a;
10
11   a__ = (a < 0 ) ? a_m : a__;
12   a_v = a__;
13   a_overPI_half =  3294199 - a_v;
14   a_v = (a_v < 1647099) ? a_v :  a_overPI_half;
15
16   //range: 1-1646264 divided by 1024
17   //(to achieve lookup-value for x: range: 0 - 1609)
18   x =  (a_v >> 10);
19   m_ = mod_1024_SP(a_v);  //modulo function with divisor 1024
20   m__ = MUL_(10000,m_); //SinglePath Multiplication Function
21   comma = (m__ >> 10); // 4 comma digit
22   x1 = x;  //(lookup-value for x1 - range: 0 - 1608)
23   x2 = x+1;
24   y1 = cos_table[x1];
25   y2 = cos_table[x2];
26   y = ppoint(x1,y1,y2,comma);
27
28   y_minus = -y;
29   y = (a__ > 1647099) ? y_minus : y;
30   return y;
31 }*/
```

In order to extend the range of input values we define the first variant in the previous listing to be the basis for all further extensions. Since the cosine function is a periodic function and since we are able to provide results for the interval [-π, π], we can assume that for each arbitray input data x, given to the algorithm, we just need to shift the value x by a multiple of 2π in order to bring it into the interval of [-π, π]. Depending on the sign of x, we need either to substract 2π from x or add it until the value gets into the interval [-π, π].

The following listing shows this adaption for an interval of [-2π, 2π] given by the lines 8-13. For greater but necessarily bounded intervals, we suggest to

insert a loop that performs the same adjustment several times and where the number of loop-iterations is constant.

Listing 4.8: Single Path Cosine - Interval: -2PI to 2PI

```
/* cosine function: interval -2PI to 2PI */
int cos_(int a)
{
  volatile int y, y_minus, y1, y2;
  volatile int x, x1, x2, x1_, rx1;
  volatile int m_, m__, comma, a_overPI_half;
  volatile int a_init = a;
  volatile int const_ = 3294199; // equals PI
  volatile int a_init1 = a - 6588397;
  volatile int a_init2 = a + 6588397;

  a_init = (a > const_) ? a_init1 : a_init;
  a_init = (a < -const_) ? a_init2 : a_init;

  volatile int a__ = a_init;
  volatile int a_m = -a_init;
  volatile int a_v;

  a__ = (a__ < 0 ) ? a_m : a__;
  a_v = a__;
  a_overPI_half =  3294199 - a_v;
  a_v = (a_v < 1647099) ? a_v :  a_overPI_half;

  x =  (a_v >> 10);
  m_ = mod_1024_SP(a_v);
  m__ = MUL_(10000,m_);
  comma = (m__ >> 10);
  x1 = x;
  x2 = x+1;
  y1 = cos_table[x1];
  y2 = cos_table[x2];
  y = ppoint(x1,y1,y2,comma);

  y_minus = -y;
  y = (a__ > 1647099) ? y_minus : y;
  return y;
}
```

4.3.7.2 Results and Measurements

Table 4.7 shows the execution time for all three different single-path Cosine implementations including the basic implementation given in Section 3.5.2 and the two extended variants that have been shown right before. At first, the table shows the execution time for the respective variant without including the arithmetic single-path functions (which is the single-path integer multiplication and division). Subsequently the table shows the respective execution time for the same implementation where the additional functions have been included. For each interval 100 random test values were used in the evaluation.

The third statement of each particular range represents an assumption that assumes the availability of proper arithmetic functions that does not cause such an overhead of execution times. First we assume that the (hardware) multiplica-

Version	*Implementation*	*Mean*	*BCET*	*WCET*	*WCET Factor*
[0, $\frac{\pi}{2}$] 1	Basic SP Cosine	292.51	291	293	1
1a	Basic SP Cosine (+ SP Func.)	922	922	922	3.15
1b	Basic SP Assumption	411	411	411	1.40
[-π, π] 2	Extended SP Cosine	348.49	347	349	1
2a	Extended SP Cosine (+ SP Func.)	978	978	978	2.80
2b	Extended SP Assumption	467	467	467	1.34
[-2π, 2π] 3	Extended SP Cosine	392.56	391	393	1
3a	Extended SP Cosine (+ SP Func.)	1022	1022	1022	2.60
3b	Extended SP Assumption	511	511	511	1.30

Table 4.7: Results of Cosine LookUp-Table Implementations

tion can be adapted very easily in a way that the execution time gets fixed to the WCET of the standard multiplication (see Table 4.5).

Furthermore we assume that it is possible to achieve a single-path software integer division that has the same WCET like the standard division (see Table 4.5). Thus we subtract the difference between the WCET of the standard division and the single-path division from an implementation that contains no single-path multiplication but the single-path division function. With that we obtain our assumption value for the execution time (see Versions 1b, 2b, 3b).

In order to evaluate the accurary of the $Cosine$ values that we achieved during our experiments we calculated on the ARM7 processor the accurate results for our the input values of all examined ranges by using the standard cos() function which is provided by the C library.

Table 4.8 shows the execution times for the standard-C cos() function and illustrates the variance of the execution time of this standard implementation. We exemplarily show the execution times for the range [0, $\frac{\pi}{2}$].

Version	*Implementation*	*Mean*	*BCET*	*WCET*
[0, $\frac{\pi}{2}$] 1	Standard C cos()	2335.69	2054	2582

Table 4.8: Results of the Standard C Cosine Function

Within our experiments, we restricted the accuracy to be single precision which appeared to be a sufficient range for our results. The single precision is usually a binary floating-point number format that occupies 4 bytes, where

23 bits of the significand appear in the memory format and where the total precision is formed by 24 bits. Thus, the accuracy is equivalent to $log_{10}(2^{24}) \approx 7.225$ decimal digits.

Version	*Implementation*	*maximal observed Deviation*	*average Deviation*
$[0, \frac{\pi}{2}]$			
1	Standard C cos()	0.0	0.0
2	Basic SP Cosine	$1 * 10^{-6}$	$2.61 * 10^{-7}$
$[-\pi, \pi]$			
1	Standard C cos()	0.0	0.0
2	Extended SP Cosine	$1 * 10^{-6}$	$4.1 * 10^{-7}$
$[-2\pi, 2\pi]$			
1	Standard C cos()	0.0	0.0
2	Extended SP Cosine	$2 * 10^{-6}$	$2.7 * 10^{-7}$

Table 4.9: Accuracy of the different Implementations

Considering the accuracy of the standard implementation (Version 1) which covers at least double precision, we can use this implementation for determining the deviations of all other implementations. Table 4.9 shows for each specific interval the deviation of each Version from the "real" value that is obtained by Version 1.

4.4 Discussion of Results

First we evaluated in our experiments algorithms that belong to the class of pure comparison-based algorithms. Concering this part of the experiments we want to point out the following observations.

- The $Find\ First$ algorithm that has been examined at first shows very clearly, that the WCET-oriented approach cannot always be applied to an algorithm wihout changing the strategy of it. In this case the strategy of the algorithm definitely had to be changed.

  In contrast to the $Find\ First$ algorithm the strategy of other ones like $Binary\ Search$ or $Bubble\ Sort$ does not need to be changed significantly. In the case of $Binary\ Sort$ the compiler behavior influences the single-path adaption much more than the slight change of the strategy for that algorithm. Besides it was also observed that for such a restriced class of algorithms the scheme for the adaption remains almost the same.

- Another interesting point regarding the compiler behavior and that class of algorithms is given by the results of the $Multi-Byte\ Counter$ algorithm. When studying the execution times one can see that the compiler translates the source code of the single-path variant much more efficiently than the traditional variant in terms of execution time. Although the source code of the traditional variant seems to comprise less complexity than the single-path variant, for the latter one a better performance is achieved. This shows that the compiler behavior is notably important for the performance of single-path code.

- When studying the results of all pure comparison-based algorithms we can conclude that the implementations on the given target hardware gain positive findings. Every algorithm could be implemented in a way that the WCET factor of their WCET-oriented respectively single-path variant is less than 1.17. This seems to be a very suitable scaling factor and allows to use the variants of these algorithms also in practice on a convenient hardware.

In this thesis we focused also on problems beyond the comparison-based class of algorithm. Therefore, we examined the well-known $Euclidean\ Greatest\ Common\ Divisor$ algorithm.

- For this algorithm we had to provide two additional numeric single-path functions which realize the required $modulo\ division$ in order to achieve the single-path variant for the $Euclidean\ GCD$ algorithm.

  Nevertheless, this variant suffers from extreme input dependencies which make the performance of the algorithm disastrous. This is due to the fact that the number of loop-iterations depends on the two input parameters that are given to the algorithm.

- This situation shows that not every algorithm is suitable for being adapted to single-path code. In such a case an alternative algorithm with another strategy is required. The $Binary\ GCD$ algorithm represents such an appropriate alterative. This algorithm is suitable for being adapted to single-path code. Its WCET factor for the adapted variant is 2.4 but it has a shorter WCET than the $Euclidean\ GCD$ algorithm.

- Regarding the source code for the single-path variant of the $Binary\ GCD$ we want to point out an observation that concerns the used compiler. According to the results for execution times and low-level code it turned out that the compiler is able to translate standard if-conditions into single-path code no matter if the branch is given as C conditional operater ("?") or not.

In order to deal with other numeric problems it was necessary to focus on basic arithmetic functions like multiplication and division. While primitive (logical) operations like shifts and comparison operations perform on the target hardware in uniform execution time (for a range of 32 bit) this is not the case for basic arithmetic functions.

- Altough it was possible to implement single-path operations for the integer multiplication and division operation we have to note that the performance, especially the one of the integer division, brings a huge overhead that can cause problems if the WCET needs to be short. The integer multiplication causes also an overhead but due to the fact that it is based on a hardware implementation it brings considerably less overhead for the execution time.

- When applying these operations to the single-path $Cosine$ implementation the results reveal the outstanding overhead for the execution times. Our assumptions on the execution times attempt to identify possible values for a scenario where adequate numeric single-path operations are available.

  Hence, there remains a huge demand for hardware implementations of mathematical functions such as multiplication and division that produce results in constant execution time and thus can substitute expensive software implementations.

  Therefore, we would even propose multiple implementations for different ranges so that one can choose the required magnitude needed for the set of numeric operations in a given algorithm. The adaption for the already presented hardware multiplication seems to be a minor problem since the hardware needs only to be forced to multiply over the whole bit-range. On the contrary the hardware implementations for integer divisions is very often not available on this kind of hardware and therefore would need to be installed completely.

Chapter 5

Summary and Conclusion

The motivation for an exact, clear and efficient WCET-analysis originates from the fact that in these days software needs to fulfil strict safety requirements and that at the same time the temporal behavior plays a crucial role. This master thesis provides an insight into strategies and solutions for several algorithms in order to make the predictability and thus the WCET analysis for these algorithms easier. Based on the already published single-path theory which treated some problems of the illustrated algorithms this thesis shows additional strategies for that approach when examining algorithmic problems of a different kind of nature.

The main strategy for reducing the complexity of WCET analysis in this work is based on the single-path approach which provides not only constant execution times, but also executions that possess a single execution path. This is the key property for simplyfying the WCET analysis. It makes path analysis unnecessary. Since the innovations of these single-path approach affect the algorithms on both, the high-level as well as the low-level code, there are different properties that need to be considered. These properties deal with requirements regarding the program code, the compiler, and the underlying target hardware and are pointed out in all detail. Regarding the target hardware it is clearly highlighted that the instruction set of the target-hardware, on the one hand, needs to comprise conditional move instructions with constant execution time, but on the other hand, requires an appropriate compiler in order to implement the desired behavior of the high-level language.

Regarding the high-level source code, the required steps for a single-path adaption are identified for different algorithms and in addition generalizations are made, in order to point out how to apply the approach to other similar algorithms.

Beyond that, a classification of algorithms is introduced, that points out that the introduced approach is only applicable within the scope of pure comparison-based algorithms. Therefore, some additional algorithmic problems are consid-

ered which prove that algorithms that comprise numeric issues cannot be solved properly using the single-path approach. Hence, the numeric problem of trigonometric functions is used in order to demonstrate how the single-path approach combined with additional techniques can yield the desired properties of uniform execution times as well as a single execution path. One of these techiques suggests that it might be useful to preprocess input-data in the form of look-up tables, in order to process the results of an algorithm in a uniform manner and constant execution time.

When implementing a $Cosine$ function for trigonometric functions that is based on a look-up tables several algorithms for basic arithmethics (including multiplication, division and modolu-division) are introduced in order to avoid the problem of execution-time variations.

Finally, the results of the experiments show that the presented comparison-based algorithms but also the algorithms implementing numeric functions satisfy the required conditions and appear to be adequate alternatives to the original variants regarding their performance and their single execution time if the limited accuracy is sufficient. Then this implementation can even compete with other implementations for the $Cosine$ since it has a considerably shorter execution time because of the use of lookup tables.

We can conclude that the impacts of WCET analysis when using the single-path approach combined with other techniques appear to be very beneficial, since the problem of expensive path analysis becomes needless. Nevertheless, there remains a demand for hardware implementations of mathematical functions such as multiplication and division that can produce results in constant execution time. If the adaption of hardware is too costly we suggest the use of a cost-efficient software implementation.

Appendix

A.1 Low Level Source Code

A.1.1 Find First Traditional Variant

With the help of the Find First example and its low-level code, we want to point out the properties which enable the single-path code. The already commented assembler code reveals that the traditional variant realizes the task with the basic mov instructions. According to the high-level code, presentend in Section 3.3.4.2, the algorithm breaks up with the main loop (label at line 10) if the branch statement in line 15 state that the searched key does not match the value that is stored in $key[i]$.

Listing A.1: Find First Tradtional

```
1  /* findfirst_trad(int *keys, int size, int key) */
2  findfirst_trad:          //R2 = key
3    PUSH     {R0,LR}
4    MVN      R3,#+0         // set R3 to NOT ZERO (-1)
5    MOV      R12,#+0        // Sets R12 to 0 - loopcounter i
6    SUB      R1,R1,#+1      // R1--  set size to size-1 (i)
7    B        ??findfirst_trad_0
8  ??findfirst_trad_1:
9    ADD      R12,R12,#+1    // inc counter variable
10 ??findfirst_trad_0:
11   CMP      R1,R12         //compare (size-1) with loop counter i
12   BLT      ??findfirst_trad_2  //branch larger then
13   LDR      LR,[R0], #+4   //loads LR from the address in R0
14   CMP      LR,R2          //compare keys[i] (LR) with key (R2)
15   BNE      ??findfirst_trad_1  //if not equal go to begin of loop
16   MOV      R3,R12         //copies current position i to R3
17 ??findfirst_trad_2:
18   MOV      R0,R3          //write R3(position) to R0
19   POP      {R12,PC}
20   END
```

A.1.2 Find First Single Path Variant

Contrary to the traditional variant, the low-level code of the following single-path implementation shows how the conditional move instruction ($MOVEQ$ in line (10)) can be used in order to implement the desired behaviour. The main loop

in this algorithm (label at line 7) runs through until all fields have been checked. The $MOVEQ$ instruction can be finally used to place the searched value. Since this instruction takes always one computation cycle, no matter if the "move" gets performed or not, we achieve a single execution path with constant execution time.

Listing A.2: Find First Single Path

```
1  /* int findfirst_sp2(int *keys, int size, int key) */
2  findfirst_sp2:              //R2 = key
3    MVN       R3,#+0           // set R3(position) to -1:
4    SUBS      R1,R1,#+1        // R1--: set variable size to size-1 (i)
5    BMI       ??findfirst_sp2_0   //quit loop if size < 0
6    ADD       R0,R0,R1, LSL #+2   //init position for array
7  ??findfirst_sp2_1:
8    LDR       R12,[R0, #+0]    //loads in R12 from the address R0+
9    CMP       R12,R2           //compare keys[i] (R12) with key (R2)
10   MOVEQ     R3,R1            // pos = i if equal,
11   SUB       R1,R1,#+1        //decrement counter i of loop
12   SUB       R0,R0,#+4        //adjust addrees for array
13   CMP       R1,#+0           //compare i with 0
14   BPL       ??findfirst_sp2_1   //if i >= 0 go to begin of loop
15 ??findfirst_sp2_0:
16   MOV       R0,R3            //write R3(position) to R0
17   MOV       PC,LR
```

A.1.3 Bubble Sort Standard Implementation

Listing A.3: Binary Search Standard Implementation

```
1  /* Bubble Sort - Traditional Impl. */
2  bubble_trad:
3    PUSH      {R4,LR}
4    CFI R14 Frame(CFA, -4)
5    CFI R4 Frame(CFA, -8)
6    CFI CFA R13+8
7    MOV       R1,#+15
8    MOV       R2,#+15
9  ??bubble_trad_0:
10   ADD       R3,R0,#+4
11   MOV       R12,R1
12 ??bubble_trad_1:
13   LDR       LR,[R3, #+0]
14   LDR       R4,[R3, #-4]
15   CMP       LR,R4
16   STRLT     R4,[R3, #+0]
17   STRLT     LR,[R3, #-4]
18   ADD       R3,R3,#+4
19   SUBS      R12,R12,#+1
20   BNE       ??bubble_trad_1
21   SUB       R1,R1,#+1
22   SUBS      R2,R2,#+1
23   BNE       ??bubble_trad_0
24   LDR       R0,[R0, #+0]
25   POP       {R4,PC}
```

A.1.4 Bubble Sort Single Path Variant

Listing A.4: GCD Binary Search Single Path Variant

```
/* Bubble Sort - Single Path Variant */
bubble_sp:
  PUSH     {R0,R4,R5,LR}
  CFI R14 Frame(CFA, -4)
  CFI R5 Frame(CFA, -8)
  CFI R4 Frame(CFA, -12)
  CFI CFA R13+16
  MOV      R1,#+15
  MOV      R2,#+15
??bubble_sp_0:
  ADD      R3,R0,#+4
  MOV      R12,R1
??bubble_sp_1:
  LDR      LR,[R3, #-4]
  LDR      R4,[R3, #+0]
  MOV      R5,R4
  CMP      R4,LR
  MOVGT    R5,LR
  STR      R5,[R3, #-4]
  CMP      R4,LR
  MOVLT    R4,LR
  STR      R4,[R3], #+4
  SUBS     R12,R12,#+1
  BNE      ??bubble_sp_1
  SUB      R1,R1,#+1
  SUBS     R2,R2,#+1
  BNE      ??bubble_sp_0
  LDR      R0,[R0, #+0]
  POP      {R3-R5,PC}
```

A.1.5 Binary Search Standard Implementation

Listing A.5: Binary Search Standard Implementation

```
/* Binary Search - Standard Implementation */
bs_trad:
  PUSH     {R0,LR}
  CFI R14 Frame(CFA, -4)
  CFI CFA R13+8
  MOV      R3,#+0
  SUB      R12,R1,#+1
??bs_trad_0:
  ADD      R1,R3,R12
  ASR      R1,R1,#+1
  LDR      LR,[R0, +R1, LSL #+2]
  CMP      LR,R2
  MOVEQ    R0,R1
  POPEQ    {R12,PC}
  CMP      LR,R2
  ADDLT    R3,R1,#+1
  SUBGE    R12,R1,#+1
  CMP      R12,R3
  BGE      ??bs_trad_0
  MVN      R0,#+0
  POP      {R12,PC}
```

A.1.6 Binary Search Single Path

Listing A.6: GCD Binary Search Single Path Variant

```
  /* Binary Search - Single Path Variant */
bs_singlep:
  PUSH     {R4,LR}
  CFI R14 Frame(CFA, -4)
  CFI R4 Frame(CFA, -8)
  CFI CFA R13+8
  MOV      R12,#+0
  SUB      LR,R1,#+1
  ASR      R3,LR,#+1
  CMP      R1,#+1
  BLT      ??bs_singlep_0
??bs_singlep_1:
  LDR      R4,[R0, +R3, LSL #+2]
  CMP      R2,R4
  SUBLT    LR,R3,#+1
  CMP      R4,R2
  ADDLT    R12,R3,#+1
  ADD      R3,R12,LR
  ASR      R3,R3,#+1
  ASR      R1,R1,#+1
  CMP      R1,#+1
  BGE      ??bs_singlep_1
??bs_singlep_0:
  LDR      R0,[R0, +R3, LSL #+2]
  CMP      R0,R2
  MVNNE    R3,#+0
  MOV      R0,R3
  POP      {R4,PC}
```

A.1.7 Multi-Byte Counter Standard Implementation

Listing A.7: Multi-Byte Counter Standard Implementation

```
/* Multi Byte Counter - Traditional Implementation */
inc_counter_trad:
  MOV      R1,#+0
  MOV      R2,R0
??inc_counter_trad_0:
  LDRB     R3,[R2, #+0]
  ADD      R3,R3,#+1
  STRB     R3,[R2, #+0]
  LDRB     R3,[R2], #+1
  CMP      R3,#+0
  BNE      ??inc_counter_trad_1
  ADD      R1,R1,#+1
  CMP      R1,#+10
  BLT      ??inc_counter_trad_0
??inc_counter_trad_1:
  LDRB     R0,[R0, #+2]
  MOV      PC,LR
```

A.1.8 Multi-Byte Counter Single Path Variant

Listing A.8: Multi-Byte Counter Single Path Variant

```
/* Multi Byte Counter - Single Path Implementation */
inc_counter_sp2:
  PUSH     {R0,LR}
  CFI R14 Frame(CFA, -4)
  CFI CFA R13+8
  MOV      R1,#+1
  MOV      R2,R0
  MOV      R3,#+10
??inc_counter_sp2_0:
  LDRB     R12,[R2, #+0]
  ADD      R12,R1,R12
  STRB     R12,[R2, #+0]
  LSLS     R12,R12,#+24
  MOVNE    R1,#+0
  ADD      R2,R2,#+1
  SUBS     R3,R3,#+1
  BNE      ??inc_counter_sp2_0
  LDRB     R0,[R0, #+2]
  POP      {R12,PC}
```

A.1.9 GCD Standard Variant (Euclid)

Listing A.9: GCD Standard - Euclid

```
/* GCD, Standard Impl */
gcd:
  PUSH     {R4,LR}
  CFI      R14 Frame(CFA, -4)
  CFI      R4 Frame(CFA, -8)
  CFI      CFA R13+8
  MOVS     R4,R0
  RSBMI    R4,R4,#+0
  CMP      R1,#+0
  BPL      ??gcd_0
  RSB      R1,R1,#+0
  B        ??gcd_0
??gcd_1:
  MOV      R4,R1
  BL       __aeabi_idivmod
??gcd_0:
  CMP      R1,#+1
  MOV      R0,R4
  BGE      ??gcd_1
  POP      {R4,PC}
```

A.1.10 GCD Single Path (Euclid)

Listing A.10: GCD Single Path - Euclid

```
/* GCD Standard - Single Path */
gcd_sp:
  PUSH     {R0,R4,R5,LR}
  CFI      R14 Frame(CFA, -4)
  CFI      R5 Frame(CFA, -8)
  CFI      R4 Frame(CFA, -12)
```

```
7   CFI       CFA R13+16
8   MOV       R4,R2
9   MOVS      R5,R0
10  RSBMI     R5,R5,#+0
11  CMP       R1,#+0
12  RSBMI     R1,R1,#+0
13  CMP       R4,#+1
14  BLT       ??gcd_sp_0
15 ??gcd_sp_1:
16  CMP       R1,#+1
17  MOVGE     R0,R5
18  MOVGE     R5,R1
19  BL        __aeabi_idivmod
20  SUBS      R4,R4,#+1
21  BNE       ??gcd_sp_1
22 ??gcd_sp_0:
23  MOV       R0,R5
24  POP       {R3-R5,PC}
```

A.1.11 GCD Binary Variant

Listing A.11: GCD Binary Variant

```
1  /* GCD, Binary Impl */
2  gcd_binary:
3    CMP       R0,#+0
4    RSBMI     R0,R0,#+0
5    CMP       R1,#+0
6    RSBMI     R1,R1,#+0
7    MOV       R2,#+0
8    CMP       R0,#+0
9    CMPNE     R1,#+0
10   BNE       ??gcd_binary_0
11   ORR       R0,R1,R0
12   MOV       PC,LR
13 ??gcd_binary_1:
14   LSR       R0,R0,#+1
15   TST       R1,#0x1
16   LSREQ     R1,R1,#+1
17   ADDEQ     R2,R2,#+1
18 ??gcd_binary_0:
19   TST       R0,#0x1
20   BEQ       ??gcd_binary_1
21 ??gcd_binary_2:
22   CMP       R1,#+0
23   BNE       ??gcd_binary_3
24   LSL       R0,R0,R2
25   MOV       PC,LR
26 ??gcd_binary_4:
27   LSR       R1,R1,#+1
28 ??gcd_binary_3:
29   TST       R1,#0x1
30   BEQ       ??gcd_binary_4
31   CMP       R0,R1
32   SUBCC     R1,R1,R0
33   BCC       ??gcd_binary_2
34   SUB       R3,R0,R1
35   MOV       R0,R1
36   MOV       R1,R3
37   B         ??gcd_binary_2
```

A.1.12 GCD Binary Single Path Variant

Listing A.12: GCD Binary Single Path Variant

```
/* GCD, Binary Single Path Impl */
gcd_binary_sp:
  PUSH     {R0,LR}
  CFI R14 Frame(CFA, -4)
  CFI CFA R13+8
  CMP      R0,#+0
  RSBMI    R0,R0,#+0
  CMP      R1,#+0
  RSBMI    R1,R1,#+0
  MOV      R2,#+0
  CMP      R0,#+0
  CMPNE    R1,#+0
  ORREQ    R1,R0,R1
  MOVEQ    R0,R1
  MOV      R3,#+31
??gcd_binary_sp_0:
  AND      R12,R0,#0x1
  EOR      R12,R12,#0x1
  AND      LR,R1,#0x1
  EOR      LR,LR,#0x1
  CMP      R12,#+0
  LSRNE    R0,R0,#+1
  CMP      LR,#+0
  LSRNE    R1,R1,#+1
  CMP      R12,#+0
  CMPNE    LR,#+0
  ADDNE    R2,R2,#+1
  SUBS     R3,R3,#+1
  BNE      ??gcd_binary_sp_0
  MOV      R3,#+60
??gcd_binary_sp_1:
  SUB      R12,R0,R1
  TST      R1,#0x1
  BEQ      ??gcd_binary_sp_2
  CMP      R12,#+0
  RSBMI    R1,R12,#+0
??gcd_binary_sp_2:
  TST      R1,#0x1
  BEQ      ??gcd_binary_sp_3
  CMP      R12,#+0
  MOVPL    R0,R1
??gcd_binary_sp_3:
  TST      R1,#0x1
  BEQ      ??gcd_binary_sp_4
  CMP      R12,#+0
  MOVPL    R1,R12
??gcd_binary_sp_4:
  LSR      R1,R1,#+1
  SUBS     R3,R3,#+1
  BNE      ??gcd_binary_sp_1
  LSL      R0,R0,R2
  POP      {R12,PC}
```

A.1.13 Basic Cosine SP Implementations: Intervall [0, $\frac{\pi}{2}$]

Listing A.13: Basic Cosine SP Implementation

```
/*
Basic Single Path Cosine Implementation
for Interval: 0 - (PI/2)
*/
cos_:
  PUSH     {LR}
  CFI      R14 Frame(CFA, -4)
  CFI      CFA R13+4
  SUB      SP,SP,#+20
  CFI      CFA R13+24
  STR      R0,[SP, #+0]
  LDR      R0,[SP, #+0]
  ASR      R0,R0,#+10
  STR      R0,[SP, #+4]
  LDR      R0,[SP, #+0]
  MOV      R2,#+0                  //start MOD_1024 procedure -
  STR      R2,[SP, #+0]            // - (contains *MUL* procedure)
  ASR      R1,R0,#+10
  STR      R1,[SP, #+0]
  LDR      R2,[SP, #+0]
  MOV      R1,#+1024
  CMP      R2,#+0                  //start MUL procedure
  RSBMI    R2,R2,#+0
  RSBMI    R1,R1,#+0
  AND      R3,R2,#0xFF
  MUL      R3,R1,R3
  MOV      R12,#+255
  AND      R12,R12,R2, ASR #+8
  MUL      R12,R1,R12
  ADD      R3,R3,R12, LSL #+8
  MOV      R12,#+255
  AND      R12,R12,R2, ASR #+16
  MUL      R12,R1,R12
  ADD      R3,R3,R12, LSL #+16
  ASR      R2,R2,#+24
  MUL      R2,R1,R2
  ADD      R1,R3,R2, LSL #+24      //end MUL procedure
  STR      R1,[SP, #+0]
  LDR      R1,[SP, #+0]
  SUB      R0,R0,R1                //end MOD_1024 procedure
  STR      R0,[SP, #+8]
  LDR      R0,[SP, #+8]
  MOV      R1,#+16
  ORR      R1,R1,#0x2700
  CMP      R0,#+0                  //start MUL procedure
  RSBMI    R0,R0,#+0
  RSBMI    R1,R1,#+0
  AND      R2,R0,#0xFF
  MUL      R2,R1,R2
  MOV      R3,#+255
  AND      R3,R3,R0, ASR #+8
  MUL      R3,R1,R3
  ADD      R2,R2,R3, LSL #+8
  MOV      R3,#+255
  AND      R3,R3,R0, ASR #+16
  MUL      R3,R1,R3
  ADD      R2,R2,R3, LSL #+16
  ASR      R0,R0,#+24
  MUL      R0,R1,R0
  ADD      R0,R2,R0, LSL #+24      //end MUL procedure
  STR      R0,[SP, #+0]
  LDR      R0,[SP, #+0]
  ASR      R0,R0,#+10
  STR      R0,[SP, #+8]
```

```
65   LDR       R1,[SP, #+4]
66   STR       R1,[SP, #+12]
67   LDR       R0,[SP, #+4]
68   ADD       R0,R0,#+1
69   STR       R0,[SP, #+0]
70   LDR       R0,??DataTable10       // cos-table
71   LDR       R1,[SP, #+12]
72   LDR       R1,[R0, +R1, LSL #+2]
73   STR       R1,[SP, #+4]
74   LDR       R1,[SP, #+0]
75   LDR       R0,[R0, +R1, LSL #+2]
76   STR       R0,[SP, #+16]
77   LDR       R3,[SP, #+8]
78   LDR       R2,[SP, #+16]
79   LDR       R1,[SP, #+4]
80   LDR       R0,[SP, #+12]
81   BL        ppoint                 //interpol function
82   STR       R0,[SP, #+0]
83   LDR       R0,[SP, #+0]
84   ADD       SP,SP,#+20
85   CFI       CFA R13+4
86   POP       {PC}
```

A.1.14 Extended Cosine SP Implementation: Intervall [-2π, 2π]

Listing A.14: Extended SP Cosine Implementation

```
1  /*
2  Basic Single Path Cosine Implementation
3  for Interval: 0 - (PI/2)
4  */
5  cos_:
6    PUSH      {LR}
7    CFI       R14 Frame(CFA, -4)
8    CFI       CFA R13+4
9    SUB       SP,SP,#+20
10   CFI       CFA R13+24
11   STR       R0,[SP, #+0]
12   LDR       R0,[SP, #+0]
13   ASR       R0,R0,#+10
14   STR       R0,[SP, #+4]
15   LDR       R0,[SP, #+0]
16   MOV       R2,#+0                 //start MOD_1024 procedure -
17   STR       R2,[SP, #+0]           // - (contains *MUL* procedure)
18   ASR       R1,R0,#+10
19   STR       R1,[SP, #+0]
20   LDR       R2,[SP, #+0]
21   MOV       R1,#+1024
22   CMP       R2,#+0                 //start MUL procedure
23   RSBMI     R2,R2,#+0
24   RSBMI     R1,R1,#+0
25   AND       R3,R2,#0xFF
26   MUL       R3,R1,R3
27   MOV       R12,#+255
28   AND       R12,R12,R2, ASR #+8
29   MUL       R12,R1,R12
30   ADD       R3,R3,R12, LSL #+8
31   MOV       R12,#+255
32   AND       R12,R12,R2, ASR #+16
33   MUL       R12,R1,R12
34   ADD       R3,R3,R12, LSL #+16
35   ASR       R2,R2,#+24
36   MUL       R2,R1,R2
37   ADD       R1,R3,R2, LSL #+24     //end MUL procedure
38   STR       R1,[SP, #+0]
```

```
39   LDR       R1,[SP, #+0]
40   SUB       R0,R0,R1                //end MOD_1024 procedure
41   STR       R0,[SP, #+8]
42   LDR       R0,[SP, #+8]
43   MOV       R1,#+16
44   ORR       R1,R1,#0x2700
45   CMP       R0,#+0                  //start MUL procedure
46   RSBMI     R0,R0,#+0
47   RSBMI     R1,R1,#+0
48   AND       R2,R0,#0xFF
49   MUL       R2,R1,R2
50   MOV       R3,#+255
51   AND       R3,R3,R0, ASR #+8
52   MUL       R3,R1,R3
53   ADD       R2,R2,R3, LSL #+8
54   MOV       R3,#+255
55   AND       R3,R3,R0, ASR #+16
56   MUL       R3,R1,R3
57   ADD       R2,R2,R3, LSL #+16
58   ASR       R0,R0,#+24
59   MUL       R0,R1,R0
60   ADD       R0,R2,R0, LSL #+24      //end MUL procedure
61   STR       R0,[SP, #+0]
62   LDR       R0,[SP, #+0]
63   ASR       R0,R0,#+10
64   STR       R0,[SP, #+8]
65   LDR       R1,[SP, #+4]
66   STR       R1,[SP, #+12]
67   LDR       R0,[SP, #+4]
68   ADD       R0,R0,#+1
69   STR       R0,[SP, #+0]
70   LDR       R0,??DataTable10        // cos-table
71   LDR       R1,[SP, #+12]
72   LDR       R1,[R0, +R1, LSL #+2]
73   STR       R1,[SP, #+4]
74   LDR       R1,[SP, #+0]
75   LDR       R0,[R0, +R1, LSL #+2]
76   STR       R0,[SP, #+16]
77   LDR       R3,[SP, #+8]
78   LDR       R2,[SP, #+16]
79   LDR       R1,[SP, #+4]
80   LDR       R0,[SP, #+12]
81   BL        ppoint                  //interpol function
82   STR       R0,[SP, #+0]
83   LDR       R0,[SP, #+0]
84   ADD       SP,SP,#+20
85   CFI       CFA R13+4
86   POP       {PC}
```

A.1.15 SP Interpolation Function for Cosine Implementation

Listing A.15: SP Interpolation Function

```
1  /* Interpolation function including MUL and DIV Operation */
2  ppoint:
3          PUSH     {R4-R7,LR}
4          CFI      R14 Frame(CFA, -4)
5          CFI      R7 Frame(CFA, -8)
6          CFI      R6 Frame(CFA, -12)
7          CFI      R5 Frame(CFA, -16)
8          CFI      R4 Frame(CFA, -20)
9          CFI      CFA R13+20
10         SUB      SP,SP,#+20
11         CFI      CFA R13+40
12         SUB      R0,R1,R2
```

```
13          STR       R0,[SP, #+12]
14          STR       R0,[SP, #+4]
15          SUB       R0,R2,R1
16          STR       R0,[SP, #+8]
17          LDR       R0,[SP, #+12]
18          CMP       R0,#+1
19          LDRGE     R2,[SP, #+4]
20          LDRLT     R2,[SP, #+8]
21          STR       R2,[SP, #+0]
22          LDR       R0,[SP, #+0]
23          CMP       R3,#+0                        //start MUL_ procedure
24          RSBMI     R3,R3,#+0
25          RSBMI     R0,R0,#+0
26          AND       R2,R3,#0xFF
27          MUL       R2,R0,R2
28          MOV       R12,#+255
29          AND       R12,R12,R3, ASR #+8
30          MUL       R12,R0,R12
31          ADD       R2,R2,R12, LSL #+8
32          MOV       R12,#+255
33          AND       R12,R12,R3, ASR #+16
34          MUL       R12,R0,R12
35          ADD       R2,R2,R12, LSL #+16
36          ASR       R3,R3,#+24
37          MUL       R3,R0,R3
38          ADD       R0,R2,R3, LSL #+24          //end MUL_ procedure
39          STR       R0,[SP, #+0]
40          LDR       R0,[SP, #+0]                //start DIV_SP procedure
41          MOV       R2,#+0
42          MOV       R3,#+0
43          MOV       R12,R2
44          MOV       LR,#+32
45          MOV       R4,#+16
46          ORR       R4,R4,#0x2700
47          RSB       R5,R4,#+0
48  ??ppoint_0:                                   //represents label of div prodecure
49          LSR       R6,R0,#+31
50          ORR       R12,R6,R12, LSL #+1
51          LSL       R0,R0,#+1
52          ADD       R6,R5,R12
53          TST       R6,#0x80000000
54          MOVEQ     R7,#+1
55          MOVNE     R7,#+0
56          CMP       R12,R4
57          MVNCS     R3,#+0
58          AND       R7,R3,R7
59          ORR       R2,R7,R2, LSL #+1
60          CMP       R7,#+0
61          MOVNE     R12,R6
62          SUBS      LR,LR,#+1
63          BNE       ??ppoint_0                  //end DIV_SP procedure
64          STR       R2,[SP, #+0]
65          LDR       R0,[SP, #+0]
66          SUB       R0,R1,R0
67          STR       R0,[SP, #+4]
68          LDR       R0,[SP, #+0]
69          ADD       R0,R0,R1
70          STR       R0,[SP, #+8]
71          LDR       R0,[SP, #+12]
72          CMP       R0,#+1
73          LDRGE     R1,[SP, #+4]
74          LDRLT     R1,[SP, #+8]
75          STR       R1,[SP, #+0]
76          LDR       R0,[SP, #+0]
77          ADD       SP,SP,#+20
78          CFI       CFA R13+20
79          POP       {R4-R7,PC}
```

A.1.16 Basic Arithmetic SP Functions

Listing A.16: Single Path Integer Multiplication

```
/* Single Path Multiplicatin for 32 x 32 bit */
MUL_:
        CMP     R1,#+0
        RSBMI   R1,R1,#+0
        RSBMI   R0,R0,#+0
        AND     R2,R1,#0xFF
        MUL     R2,R0,R2
        MOV     R3,#+255
        AND     R3,R3,R1, ASR #+8
        MUL     R3,R0,R3
        ADD     R2,R2,R3, LSL #+8
        MOV     R3,#+255
        AND     R3,R3,R1, ASR #+16
        MUL     R3,R0,R3
        ADD     R2,R2,R3, LSL #+16
        ASR     R1,R1,#+24
        MUL     R1,R0,R1
        ADD     R0,R2,R1, LSL #+24
        MOV     PC,LR
```

Listing A.17: Single Path Integer Division

```
udiv_SP:
        PUSH    {R0,R4,R5,LR}
        CFI R14 Frame(CFA, -4)
        CFI R5 Frame(CFA, -8)
        CFI R4 Frame(CFA, -12)
        CFI CFA R13+16
        MOV     R3,#+0
        MOV     R12,#+0
        MOV     R2,R3
        MOV     LR,#+32
??udiv_SP_0:
        LSR     R4,R0,#+31
        ORR     R12,R4,R12, LSL #+1
        LSL     R0,R0,#+1
        SUB     R4,R12,R1
        TST     R4,#0x80000000
        MOVEQ   R5,#+1
        MOVNE   R5,#+0
        CMP     R12,R1
        MVNCS   R3,#+0
        AND     R5,R3,R5
        ORR     R2,R5,R2, LSL #+1
        CMP     R5,#+0
        MOVNE   R12,R4
        SUBS    LR,LR,#+1
        BNE     ??udiv_SP_0
        MOV     R0,R2
        POP     {R3-R5,PC}
        CFI EndBlock cfiBlock3
```

Listing A.18: Single Path Mod 1024 Division

```
/*
MODULO Div - function with fixed divisor 1024:
contains MUL procedure
*/
mod_1024_SP:
        SUB     SP,SP,#+8
        CFI     CFA R13+8
        MOV     R2,#+0
```

```
9           STR       R2,[SP, #+0]
10          ASR       R1,R0,#+10
11          STR       R1,[SP, #+0]
12          LDR       R2,[SP, #+0]
13          MOV       R1,#+1024
14          CMP       R2,#+0                  //start MUL
15          RSBMI     R2,R2,#+0
16          RSBMI     R1,R1,#+0
17          AND       R3,R2,#0xFF
18          MUL       R3,R1,R3
19          MOV       R12,#+255
20          AND       R12,R12,R2, ASR #+8
21          MUL       R12,R1,R12
22          ADD       R3,R3,R12, LSL #+8
23          MOV       R12,#+255
24          AND       R12,R12,R2, ASR #+16
25          MUL       R12,R1,R12
26          ADD       R3,R3,R12, LSL #+16
27          ASR       R2,R2,#+24
28          MUL       R2,R1,R2
29          ADD       R1,R3,R2, LSL #+24    //end  MUL
30          STR       R1,[SP, #+0]
31          LDR       R1,[SP, #+0]
32          SUB       R0,R0,R1
33          ADD       SP,SP,#+8
34          CFI       CFA R13+0
35          MOV       PC,LR
```

A.2 Ian Kaplan - Radix two Integer Division Algorithm

Listing A.19: Radix two Interger Division

```
1  /* Copyright (c) - Ian Kaplan, October 1996  */
2
3  void unsigned_divide(unsigned int dividend,
4                       unsigned int divisor,
5                       unsigned int &quotient,
6                       unsigned int &remainder )
7  {
8    unsigned int t, num_bits;
9    unsigned int q, bit, d;
10   int i;
11   remainder = 0;
12   quotient = 0;
13   if (divisor == 0)
14     return;
15   if (divisor > dividend) {
16     remainder = dividend;
17     return;
18   }
19   if (divisor == dividend) {
20     quotient = 1;
21     return;
22   }
23   num_bits = 32;
24   while (remainder < divisor) {
25     bit = (dividend & 0x80000000) >> 31;
26     remainder = (remainder << 1) | bit;
27     d = dividend;
28     dividend = dividend << 1;
29     num_bits--;
30   }
31   /* The loop, above, always goes one iteration too far.
32      To avoid inserting an "if" statement inside the loop
33      the last iteration is simply reversed. */
34   dividend = d;
35   remainder = remainder >> 1;
36   num_bits++;
37   for (i = 0; i < num_bits; i++) {
38     bit = (dividend & 0x80000000) >> 31;
39     remainder = (remainder << 1) | bit;
40     t = remainder - divisor;
41     q = !((t & 0x80000000) >> 31);
42     dividend = dividend << 1;
43     quotient = (quotient << 1) | q;
44     if (q) {
45        remainder = t;
46      }
47   }
48 }
```

Abbreviations

ACET	Average Case Execution Time
BCET	Best Case Execution Time
CFG	Control-Flow Graph
ET	Event-Triggered
FPU	Floating Point Unit
GCD	Greatest Common Divisor
I/O	Input / Output
ISA	Instruction Set Architecture
NOP	No Operation
TT	Time-Triggered
TTP	Time-Triggered Protocol
WCET	Worst Case Execution Time

Bibliography

[1] Wilhelm R. Engblom J. Ermedahl A. Holsti N. Thesing S. Whalley D. Bernat G. Ferdinand C. Heckmann R. Mitra T. Mueller F. Puaut I. Puschner P. Staschulat J. Stenstroem P. The worst-case execution-time problem Ů Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems; 36; 1539-9087*, 2008.

[2] Reineke J. Wachter B. Thesing S. Wilhelm R. Polian I. Eisinger J. and Becker. A definition and classification of timing anomalies. *WCET 2006*, 2006.

[3] Puschner P. and Burns A. Writing Temporally Predictable Code. *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002);85; 1530-1443*, 2002.

[4] Puschner P. Algorithms for Dependable Hard Real-Time Systems. *Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03), 26-31*, 2003.

[5] Allen J. Kennedy K. Porterfield C. and Warren J. Conversion of control dependence to data dependence. *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 177-189, NY, USA, 1983. ACM.*, 1983.

[6] Puschner P. Evaluation of the Single-Path Approach and WCET-Oriented Programming. *Technical University Vienna, Department of Computer Science*, 2007.

[7] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[8] Thomas Lundqvist and Per Stenstroem. Timing anomalies in dynamically scheduled microprocessors. *IEEE Computer Society:20th IEEE Real-Time Systems Symposium*, 1999.

[9] Heckmann R. Langenbach M. Thesing S. and Wilhelm R. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems 91; 7; 1038Ű1054*, 2003.

[10] Reinhard Wilhelm. *Determining Bounds on Execution Times. In Handbook on Embedded Systems*. Ed. CRC Press, 2005.

[11] Kadlec A. and Kirner R. On the Difficulty of Building a Precise Timing Model for Real-Time Programming. *14th Colloquium of Programminglanguages and Basics of Programming, pages 99-105*, 2007.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufman Publishers, 3rd Edition, 2003.

[13] Kirner R. and Puschner P. Classification of wcet analysis techniques. *Technical report, Technical University Vienna, Department of Computer Science*, 2002.

[14] John L. Hennessy and David A. Patterson. *Computer Organization and Design*. Morgan Kaufman Publishers, 2nd Edition, 1998.

[15] ATMEL. Arm7 tdmi technical reference manual.

[16] Puschner P. Transforming Execution-Time Boundable Code into Temporally Predictable Code. *IFIP Conference Proceedings; Vol. 219, Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems, pages 163 - 172*, 2002.

[17] Vitit Kantabutra. On hardware for computing exponential and trigonometric functions. *IEEE Computer Society Washington, DC, USA, Vol 45, 328-329*, 1996.

[18] Kirner R. Groessing M. and Puschner P. Comparing WCET and Resource Demands of Trigonometric Functions Implemented as Iterative Calculations vs. Table-Lookup. *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[19] Joseph J. F. Cavanagh. *Digital Computer Arithmetic*. McGraw-Hill, Inc, 1983.