Faculty of Economic Sciences, Communication and IT
Department of Computer Science

Rikard Boström Lars-Olof Moilanen

Capacity profiling modeling for baseband applications

Degree Project of 30 credit points
Master of Science in Information Technology

Date/Term:	2009-01-15
Supervisor:	Thijs Holleboom
Examiner:	Donald Ross
Serial Number:	E2009:03

Karlstads Universitet 651 88 Karlstad

Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Capacity profiling modeling for baseband applications

Rikard Boström Lars-Olof Moilanen

© 2009 The author and Karlstad University

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Rikard Boström

Lars-Olof Moilanen

Approved, 2009-01-15

Advisor: Thijs Holleboom

Examiner: Donald Ross

Abstract

Real-time systems are systems which must produce a result within a given time frame. A result given outside of this time frame is as useless as not delivering any result at all. It is therefore essential to verify that real-time systems fulfill their timing requirements. A model of the system can facilitate the verification process. This thesis investigates two possible methods for modeling a real-time system with respect to CPU-utilization and latency of the different components in the system. The two methods are evaluated and one method is chosen for implementation.

The studied system is the decoder of a Wideband Code Division Multiple Access (WCDMA) system which utilizes a real-time operating called system Operating System Embedded compact kernel (OSEck). The methodology of analyzing the system and different ways of obtaining measurements to base the model upon will be described. The model was implemented using the simulation library VirtualTime, which contains a model of the previously mentioned operating system. Much work was spent acquiring input for the model, since the quality of the model depends largely on the quality of the analysis work. The model created contains two of the studied systems main components.

This thesis identifies thorough system knowledge and efficient profiling methods as the key success factors when creating models of real-time systems.

Acknowledgements

We would like to thank our supervisors Mikael Carlsson, Per Olsson and Tor Suneson at Tieto for their guidance during this thesis. We also want to thank everyone else at Tieto who has assisted with environments, testing and the version handling system. Finally, a big thank you goes to our supervisor Thijs Jan Holleboom at Karlstad University.

Contents

1 Introduction **1**

1.1 The need for a model 1

1.2 The studied system 2

1.3 Goal of thesis 3

2 Background **5**

2.1 Introduction 5

2.2 Real-time systems 5

2.2.1 Hard real-time system 6

2.2.2 Soft real-time system 6

2.3 Verification and analysis of real-time systems 6

2.3.1 Measuring on the actual system 7

2.3.2 Creating a model of the system 9

2.4 The studied real-time system 9

2.4.1 Overview of system components 11

2.4.2 The user data processing chain 14

2.5 Summary 16

3 Feasibility study **17**

3.1 Introduction 17

3.2 The two models 17
3.2.1 Model based on target code 18
3.2.2 Abstract model 19
3.3 Requirements on the model 22
3.3.1 Information output 23
3.3.2 Accuracy 24
3.3.3 Verification 25
3.3.4 Input 25
3.3.5 Output format 26
3.3.6 Limited cost of modeling current software 27
3.3.7 Limited cost of modeling current hardware 28
3.3.8 Limited cost of modeling changes in software 29
3.3.9 Limited cost of modeling changes in hardware 30
3.4 Use cases 31
3.4.1 Modeling new application features in early project phase 31
3.4.2 Identifying worst case 32
3.4.3 Identify bottlenecks in the system 33
3.5 Prioritized requirements 34
3.5.1 Accuracy 34
3.5.2 Identifying worst case 35
3.5.3 Model changes in the software 35
3.6 Model choice 35
3.7 Summary 36

4 Model creation methodology 37
4.1 Introduction 37
4.2 Methodology overview 37
4.3 Input sources for the model creation process 38

4.3.1 The instruction set simulator used for profiling the code 39
4.3.2 The target event logs 40
4.4 VirtualTime 40
4.4.1 VirtualTime entities 41
4.5 Use of VirtualTime in the model 45
4.5.1 Modeling of processes 45
4.5.2 Modeling of hardware 45
4.5.3 Modeling of software functions 45
4.6 Model limitations 46
4.6.1 Only user data of type EDCH 46
4.6.2 Only user data with 2 ms TTI 47
4.6.3 Omitted control plane 47
4.6.4 Modeling limited to two components 47
4.6.5 Omitted retransmissions 47
4.6.6 Little conditional execution 48
4.6.7 Only one user 48
4.6.8 Summary of limitations 48
4.7 Modeling the different parts 48
4.7.1 Application scheduler 49
4.7.2 Turbo Decoder Peripheral 58
4.8 Chapter summary 60

5 Verification **61**
5.1 Introduction 61
5.2 Comparison between target and the instruction set simulator 61
5.2.1 The instruction set simulator code compiled with debug and no compile time optimizations 62

5.2.2 The instruction set simulator code compiled with debug and compile time optimizations . . . 63
5.2.3 The instruction set simulator code compiled without debug, with compile time optimizations . . . 64
5.2.4 Discussion about the comparisons between target and the instruction set simulator . . . 65
5.2.5 Addressing the found issues . . . 67
5.2.6 Final comparison . . . 68
5.3 Comparison between Model and target . . . 69
5.3.1 Further discussion about remaining deviations . . . 70

6 VirtualTime implementation **73**
6.1 Introduction . . . 73
6.2 Limitations . . . 73
6.3 Overview . . . 74
6.4 Implementation of each component . . . 74
6.4.1 User Data . . . 75
6.4.2 Frame buffer . . . 76
6.4.3 Frame buffer interrupt service routine . . . 76
6.4.4 DMA . . . 77
6.4.5 DMA interrupt process . . . 77
6.4.6 TDP . . . 78
6.4.7 Application Scheduler . . . 79

7 Discussion **81**
7.1 Introduction . . . 81
7.2 The model choice . . . 81
7.3 Simulation library . . . 82

7.4 Important aspects when modeling 82
7.4.1 System knowledge 82
7.4.2 Profiling 83
7.5 Model future work 83

8 Conclusion **85**

References **87**

Acronyms **89**

List of Figures

1.1 *A figure illustrating the studied systems real-time requirements* 2

2.1 *A figure depicting the decoders placement in the signal processing chain.* . . 10

2.2 *Flow chart illustrating the signal processing steps involved in processing of 2ms EDCH user data in the uplink direction according to 3GPP TS 25.212 version 6.4.0 Release 6.* 12

2.3 *Sequence diagram illustrating a simplified view of the processing of 2ms EDCH user data from an implementation perspective.* 15

3.1 *A simple VirtualTime code example, ping-pong.* 21

4.1 *A code snippet illustrating how to receive a signal of some specific type(s) in VirtualTime.* 44

4.2 *A code snippet returning the amount of cycles consumed by a bubble sort implementation.* 46

4.3 *A graph illustrating the correlation between $y =$ the cycles consumed by the second deinterleaving step and $x =$ the length of the data being processed.* . . 52

4.4 *A graph illustrating the correlation between $x = [numberOfCodeBlocks] * ([codeBlockSize] + 4)$ and $y =$the amount of cycles consumed by the rate dematching step.* 53

4.5 *A graph illustrating the correlation between the cycles consumed by the limit and segment step (y) and a mathematical function composed of the number of code blocks and the code block size (x)* 54
4.6 *A graph illustrating the correlation between the cycles consumed by the function calcTdpParams (z) and a mathematical function composed of the number of code blocks (x) and the code block size (y)* 56
4.7 *A graph illustrating the correlation between* $y =$ *the amount of clock cycles consumed by pre-TDP, non-profiled functions and* $x =$ *the number of symbols being processed.* 57
4.8 *A graph illustrating the connection between* $y =$ *the delay (amount of clock cycles) from calling TDP to receiving completion signal and* $x = [codeBlockSize] * [numberOfCodeBlocks]$. 59

6.1 *General component design in VirtualTime* 75
6.2 *A figure illustrating the memory/cache hierarchy of the studied system.* . . 78

List of Tables

4.1 *The amount of cycles used per symbol for different values of the SymbolType parameter* 51

4.2 *Formulas for calculating cycle consumption for different functions and components* 60

5.1 *Comparison between the instruction set simulator and target for test case 1, instruction set simulator code compiled with debug and no compile time optimizations* 62

5.2 *Comparison between the instruction set simulator and target for test case 2, instruction set simulator code compiled with debug and no compile time optimizations* 62

5.3 *Comparison between the instruction set simulator and target for test case 1, instruction set simulator code compiled with debug and compile time optimizations (O=3)* 63

5.4 *Comparison between the instruction set simulator and target for test case 2, instruction set simulator code compiled with debug and compile time optimizations (O=3)* 64

5.5 *Comparison between the instruction set simulator and target for test case 1, instruction set simulator code compiled without debug, with compile time optimizations (O=3)* 64

5.6 *Comparison between the instruction set simulator and target for test case 2, instruction set simulator code compiled without debug, with compile time optimizations (O=3)* 65
5.7 *Final comparison between the instruction set simulator and after addressing the found issues for test case 1.* 68
5.8 *Final comparison between the instruction set simulator and target after addressing the found issues for test case 2.* 68
5.9 *Comparison between target and model for test case 1.* 69
5.10 *Final comparison between target and model for test case 2.* 70

Chapter 1

Introduction

1.1 The need for a model

In a real-time system there are constraints on the latency of processing input data and producing output, i.e. a result. If the system is under heavy load, and hence has very high resource utilization, care must be taken when adding new features. An increased computation time introduced somewhere in the code could affect the whole system in terms of scheduling and interrupt handling. Assume time has been spent implementing a new feature which during testing pushes the system over the limit, i.e. the system no longer meets its deadlines. When doing such a finding late in the development process, it might be a costly procedure to spend even more time to redesign the new feature or other parts of the system [17, 10].

If the problem instead could be detected earlier, this risk can be mitigated. This could be done by classifying the feature request as infeasible or give it a larger work estimate, and hence avoiding spending more time and money than reasonable. Creating an abstraction of the real-time system, a model, that allows prototyping of new features would make this possible.

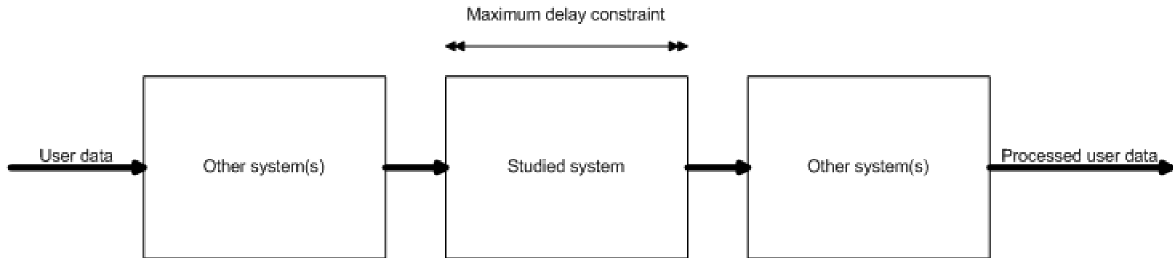

Figure 1.1: *A figure illustrating the studied systems real-time requirements*

1.2 The studied system

The system studied in this thesis is a real-time system that is a smaller part of a larger telecommunications system. The system is a baseband application, which means it performs signal processing on a baseband signal, i.e. a signal that has been downmixed from the carrier frequency to the original frequency. The function of baseband is to output data identical to the data going into the transmitter on the sender side by applying signal processing algorithms on the baseband signal. Some of the algorithms correct errors induced by the transmission.

The system has constraints on the maximum delay it may introduce in the chain of processing steps taken place in the telecommunication system, i.e. a limit for the maximum time it may take from the arrival of the user data and the delivering of the post-processed ditto (see figure 1.1).

The system has requirements on handling many concurrent users. When this happens, interrupts and context switches occur constantly and the real-time aspects of the system is really being put to test.

Previously, the studied system was validated only by doing measurements on it, which, according to the engineers working with the system, might not really identify the Worst Case Execution Times (WCETs) of tasks. This is also supported by Andreas Ermedahl's Ph.D. thesis [8]:

> "The traditional way to determine the timing of a program is by measurements, also known as dynamic timing analysis. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [Ive98, Ste02]. The methodology is basically the same for all approaches: run the program many times and try different potentially "really bad" input values to provoke the WCET. This is time-consuming and difficult work, which does not always give results which can be guaranteed."

As he points out there is no guarantee that the worst case execution times can be found by measuring, and incorrect worst case execution time estimates could be used as input to the timing analysis method, resulting in an incorrect result. Since an accurate timing analysis can be highly valuable, the uncertainty of the measurement method is one of the main reasons for looking at a supplementary validation method.

1.3 Goal of thesis

This thesis investigates the creation of a model that helps address the issues described in section 1.1, namely identifying the worst case and also enabling early estimation of resource utilization of new features without actually implementing them. Two different approaches for creating such a model are studied and discussed. A choice of one of these approaches is made, and the different aspects and problems found by using that approach when attempting to create such a model will be discussed.

Chapter 2

Background

2.1 Introduction

In this chapter relevant background information needed to understand this thesis will be given. It is assumed that the reader of this thesis is familiar with the fundamentals of computer science. The term real-time system and the two main types of real-time systems, hard and soft, will be defined. An explanation why verification and analysis of real-time systems are necessary will be given, and existing methods for achieving this will be described. The studied system will also be presented.

2.2 Real-time systems

A real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the time instance at which these results are produced [9].

2.2.1 Hard real-time system

Hard real-time systems are systems that must meet their temporal specification in all anticipated load and fault scenarios [9], otherwise catastrophic consequences could occur, e.g. damage to equipment, personal injury or even death. An example of a hard real-time system is the engine control system of a car, if the system should fail to meet its deadline it may cause the engine to fail or get damaged.

2.2.2 Soft real-time system

A soft real-time system is less restrictive than a hard one, "simply providing that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes" [14].

In a soft real-time system a missed deadline does not lead to catastrophic scenarios, it is more likely that the performance of the system is reduced which can indeed be irritating for a user, but not dangerous. An example of a soft real-time system could be a decoder processing streaming video. A missed deadline may result in a skipped frame or stuttering of the video stream. Irritating but hopefully not dangerous. The system in this thesis is a soft real-time system, but the real-time performance of the system can be considered a part of the functionality. It is hence very important that the system fulfills its deadlines even if failing to do so does not lead to any physical damage.

2.3 Verification and analysis of real-time systems

Because of the timing constraints placed on real-time systems, there is obviously a need for analyzing and verifying that real-time systems fulfill them, as Andersson et. al. state in [5]:

> "If the software system has real-time requirements, it is of vital importance

that the system is analyzable with respect to timing related properties, e.g. deadlines."

When verifying a real-time system the goal is to simply provoke the system with the worst-case of input data, and see if it still manages to fulfill its timing requirements. The worst-case of input data is the data that the system can be exposed to which gives the system most problems fulfilling its deadlines. The difficulty of identifying input data representing the worst case is an important problem and is further discussed in section 3.4.2. Also, an analysis of the systems timing properties can help indicate where there are bottlenecks and needs for optimizations.

The two methods for verifying real-time systems will be briefly described. The first method consists of performing measurements on the actual system, the second is to create a model that is an abstraction of the system and that captures the timing aspects of the system. While performing measurements on the actual system seems straight forward it has some limitations, which a model of the system has not. This is why exploring how to create a model of a real-time system is the main motivation for this thesis.

2.3.1 Measuring on the actual system

One method of verifying that a system fulfills its real-time requirements is to make measurements on the actual system. For the system studied in this thesis this is realized by using a logic analyzer in conjunction with the insertion of code segments at selected places in the code that outputs data during execution of system.

For the purposes of this thesis a logic analyzer is an electronic instrument which record the signals being sent in digital circuits, logging the observed values to a file for later analysis. See [18] for a brief summary on logic analyzers.

By having the logic analyzer listening on the external memory bus, data written to the external memory can be caught and saved to a file without interfering with the system. As the logic analyzer does not interfere with the system, it avoids the so-called probe ef-

fect. The probe effect [19] means that the measurement itself affects the result, because additional resources are needed for the output of logging information. However, it is not physically possible to use the logic analyzer for the internal memory of the Central Processing Unit (CPU) or other internal components because the pins are unavailable. Therefore, the information hidden inside these internal components must somehow explicitly be written to the external memory, where it can be recorded by the logic analyzer. When inserting code segments to cope with the physical limitations of using the logic analyzer, one has to be aware of the probe effect. The probe effect occur since the new code to exploit the data otherwise trapped in the internal unavailable components uses resources and introduces delay[13]. It might therefore be preferable to always have the probes in the code, even if the output is not used [4]. In the studied system the resource utilization of the system is already high, and insertion of code segments into the production code is not considered to be an option.

As previously mentioned, it is of interest to find test data which represents the worst case, that is the one that results in the longest execution time that the system may be subjected to, and running that test data through the system. The log files from the measurements are then analyzed.

The advantage of this method, i.e. measuring on the actual system, is that the system itself serves as a model, which makes it very accurate. One of the major disadvantages of this method is that it is hard to find test data that represents a worst case. One can also not be certain that the actual worst case has been identified [8]. It might be the worst case observed so far, but there could still be other test data that provokes the system further. It is thus hard, if not impossible, to verify that the worst case is actually tested. Another problem with this method is that the complexity of the system makes it hard to analyze the results of the measurements.

While measuring on the actual system yields accurate results, the work involved in creating input data is substantial. Furthermore the test cases need to be run for a sub-

stantial amount of time. This, together with the requirement of 100% correct code, i.e. a completely working system from a functional point of view, makes it unsuitable for rapid testing of new designs, new features and new hardware characteristics.

2.3.2 Creating a model of the system

Instead of measuring on the actual system, it is possible to create a model of it where the timing aspects of the different parts of the system are preserved. Investigating the creation of such a model is the goal of this thesis. The model is an abstraction that captures the important aspects of the actual system, which in this case have been previously identified as CPU-utilization and delay of the different components. A model has the benefit of only showing what is of interest, i.e. latency contributions, and hence reducing the complexity, which makes the analysis process easier. Other benefits, such as the possibility to simulate new hardware characteristics and the ease of modeling changes in the code at an early stage, have also been identified. One key issue when creating a model of a real-time system is if the level of accuracy required can be reached. Another key issue is to keep the model updated as the system evolves.

This thesis aims to investigate the creation of a model of the system to help prototyping new features and simulate new hardware and designs. It should also facilitate finding the worst possible combination of input data the system may be subjected to, and help verifying the fulfillment of the timing constraints and ease the identification of bottlenecks in the system.

2.4 The studied real-time system

The studied system is a decoder, as defined by the 3rd Generation Partnership Project (3GPP)[1], in a WCDMA system. WCDMA is the technology used to implement and realize Universal Mobile Telecommunications System (UMTS) [15]. The decoder is the last

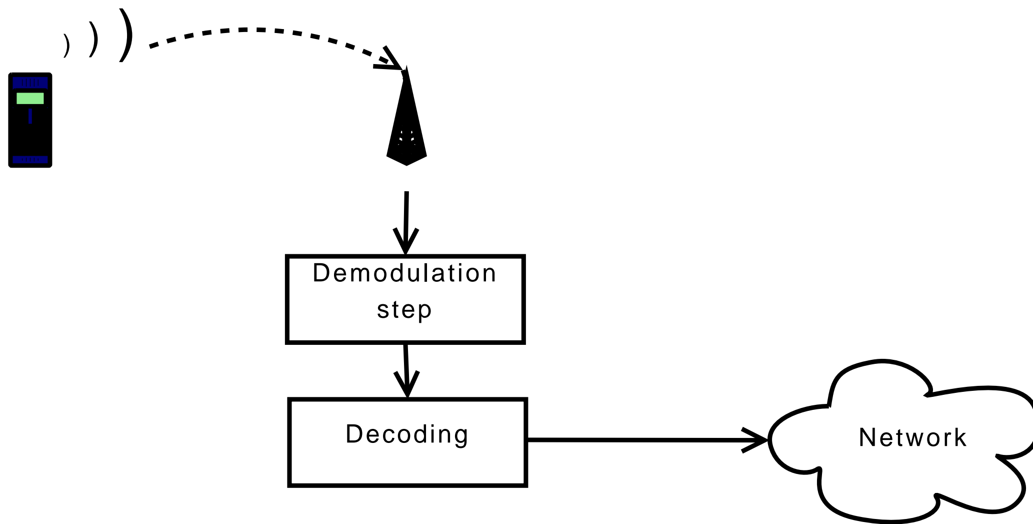

Figure 2.1: *A figure depicting the decoders placement in the signal processing chain.*

unit of the user plane layer 1 processing chain (see figure 2.1) and is responsible for doing signal processing and packaging of user data. The Radio Network Controller (RNC) then operates on the processed flows. The user data received by the decoder contains the actual user data sent by the User Equipment (UE), e.g. a cell phone, but also extra information used for detecting and correcting errors in the received data. The extra information is used by the decoder to make sure that the data leaving the system is correct.

The system runs on a CPU, which makes use of a coprocessor and communicates with a number of other hardware devices. Most of the studied application is written in C, with some minor parts in assembler. A real-time operating system, OSEck[11], is used. There are constraints on the maximum time it may take from receiving an interrupt signaling that new user data is available, to the time the processed user data leaves the system. Even though failing to meet its timing requirements does not result in any immediate catastrophal event, the system constraints are somewhat harder than for the general soft real-time system described in section 2.2.2.

The signal processing steps which are to be applied to the user data before they are

sent from the user equipment are described in release 6 of the 3GPP standard[2], the steps which are applied in the uplink direction are illustrated in figure 2.2.

2.4.1 Overview of system components

This thesis was limited to study the processing of one specific type of user data (see section 4.6 about the model's limitations), namely Enhanced Dedicated Channel (EDCH) traffic with a 2 ms Transmission Time Interval (TTI). Seven components were identified to be involved in the processing of this type of user data. These components will now be presented.

Frame buffer

The user data from the demodulation step is stored in the Frame Buffer (FB) before the decoder fetches it. When new user data arrives to the FB from the demodulation the FB generates an interrupt acknowledging the decoder that new user data is available for processing. This interrupt causes the FB Interupt Service Routine (ISR) to launch.

Frame buffer Interrupt Service Routine

The FB ISR runs whenever the FB contains new data which is to be decoded. The user data is copied from FB to the internal memory by using the Direct Memory Access (DMA) controller. A DMA job is started and the ISR exits.

DMA controller

The DMA controller is a hardware resource responsible for transferring data between the different hardware components and the internal and external memory. There are two main types of DMA usage, implicit and explicit. Implicit usage is when fetching data from the external memory, which must go through the DMA. The explicit usage is that code segments are used for explicitly moving data from one location to another, e.g. between

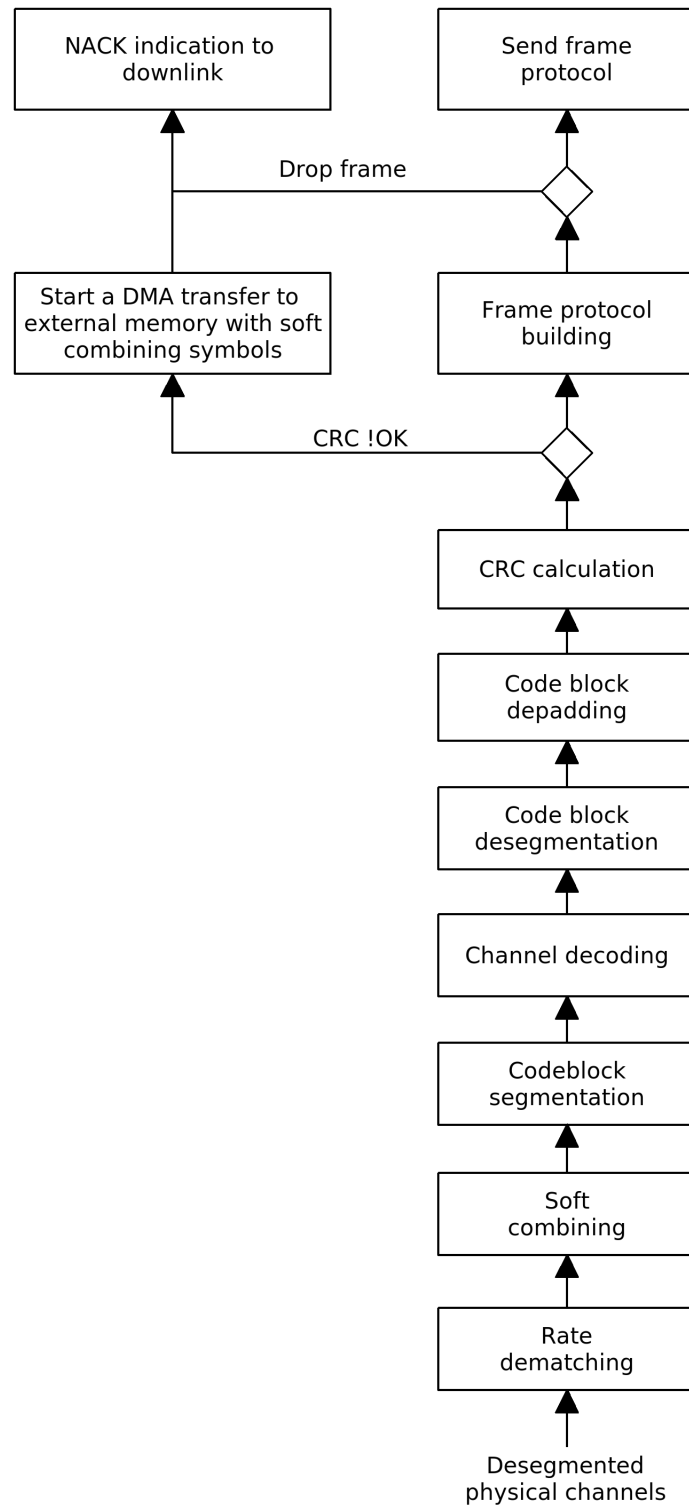

Figure 2.2: *Flow chart illustrating the signal processing steps involved in processing of 2ms EDCH user data in the uplink direction according to 3GPP TS 25.212 version 6.4.0 Release 6.*

two different hardware components. When an explicit DMA job has finished, an interrupt can be generated which is caught by the DMA interrupt process.

DMA Interrupt Process

The DMA Interrupt Process runs with high priority and is swapped in when a DMA job is finished because of the interrupt generated by the DMA controller. The process then sends a signal to the application scheduler process with information on where the new data can be accessed.

Application Scheduler Process

The application scheduler is a process that roughly does two things: Signal processing (according to figure 2.2) and packaging of the user data. The channel decoding step is performed by the Turbo Decoder Peripheral (TDP). The application scheduler processes the user data to the point where it is ready to start the TDP. At this point the data which is to be decoded are input to the TDP by using the DMA controller. When the TDP is finished, the user data is packaged and sent out of the decoder. The application scheduler process consists mainly of a big loop which for every iteration gets a signal from its signal queue and takes proper action depending on the signal, e.g. do signal processing or user data packaging (called frame protocol building).

Turbo Decoder Peripheral

TDP is short for Turbo Decoder Peripheral and is a hardware which decodes turbo encoded data. It is invoked by a DMA job, and when finished the DMA controller generates an interrupt which triggers the DMA interrupt process.

Frame Protocol Driver

The Frame Protocol (FP) driver is an external process (not owned by the decoder) to which the finished packaged user data is written. It is not studied in this thesis.

2.4.2 The user data processing chain

A description on how the different components interact will now be given. The flow time window is set from when the FB sends an interrupt acknowledging that new user data is available to when the finished and packaged user data leaves the decoder. The flow is depicted in figure 2.3.

When user data has arrived to the FB, the FB generates an interrupt to acknowledge the decoder that there is user data available. The interrupt launches the FB ISR which starts a DMA job for transferring the user data from FB to the internal memory. The DMA controller copies the user data and sends an interrupt that is caught by the DMA interrupt process. The DMA interrupt process sends an acknowledgement to the FB, saying that it is now allowed to send another interrupt. It also sends a signal to the application scheduler process with information that user data now resides in the internal memory and is now ready for processing. The application scheduler process does a large part of the signal processing and sets up a TDP job by calculating some parameters and invoking a number of DMA jobs. The DMA starts the TDP and makes sure it gets the user data and the parameters needed for correct processing. The TDP processes the user data, and when finished, it makes the DMA aware of this. The DMA controller generates an interrupt which is caught by the DMA interrupt process. The DMA interrupt sends a signal to the application scheduler with information that the user data is now processed and ready for packaging. The application scheduler then packages the user data and sends it out of the system.

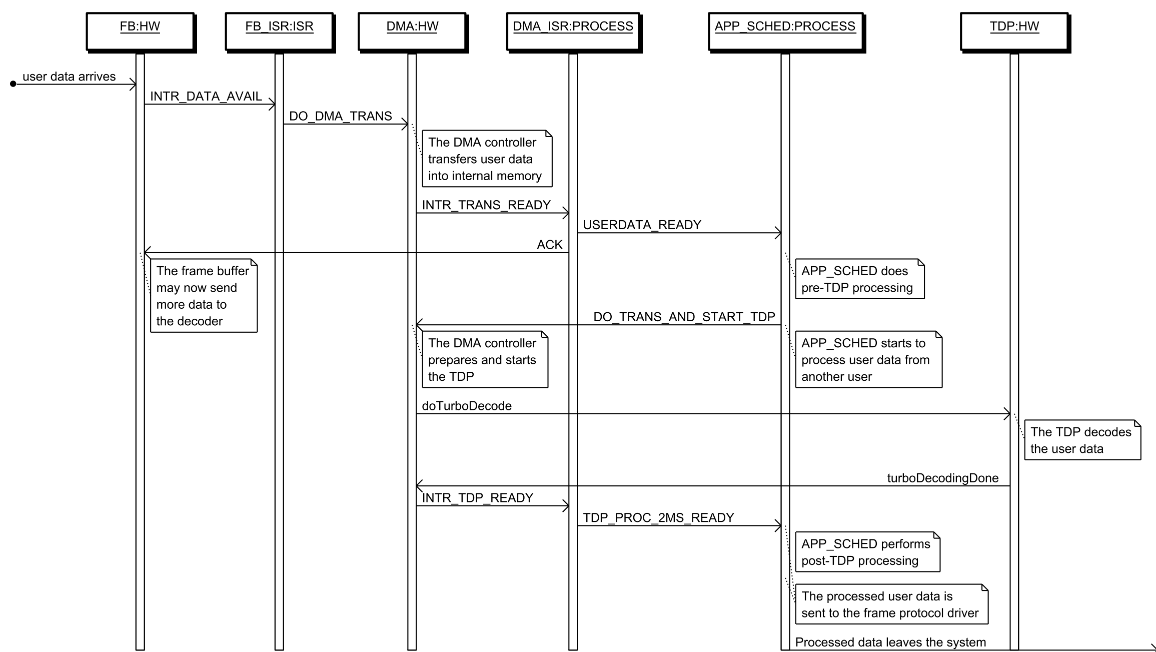

Figure 2.3: *Sequence diagram illustrating a simplified view of the processing of 2ms EDCH user data from an implementation perspective.*

2.5 Summary

Real-time systems are systems which have constraints on the processing time of input data. There are hard and soft real-time systems which differ in the importance of fulfilling these constraints. The system studied in this thesis is classified as a soft real-time system.

It is important to verify the fulfillment of the constraints placed upon real-time systems. Two alternative methods of doing this are discussed, measuring on the actual system or creating a model of it. This thesis investigates the latter approach which has benefits compared to the first approach. A model could help finding the worst case and predict the impact of new features.

Chapter 3

Feasibility study

3.1 Introduction

Two different approaches for creating a model of the system had been identified prior to this thesis. The first one runs the production code in an instruction set simulator, while the other runs an abstraction of the system in a simulation environment. The feasibility study was the process of evaluating these two approaches and selecting which of them to use. The models were considered based on a set of requirements and everyday usages. In this chapter, the requirements will be described and motivated. The two approaches will be described along with discussions on if and how they can fulfill the requirements. Typical everyday usages of the model, and their potential of fulfillment in each model, will also be presented. The usages can be seen as a concretization of one or more requirements and how the model is applied in commonly occurring scenarios.

3.2 The two models

In this section the two different approaches, or models, will be briefly described together with the initial thoughts concerning benefits, drawbacks and suitability. The models will

then be further explained in conjunction with the requirements in section 3.3, where a deeper analysis of benefits and drawbacks of each model is carried out. This analysis is the basis for the decision of which model will be used.

3.2.1 Model based on target code

This model is based on the target code running in an instruction set simulator. For the purposes of this thesis, target code is defined as the actual production code that runs on the real hardware. The application itself, the operating system and modeled peripheral hardware are compiled into a binary file which is loaded in the simulator. Measurements can then be made by using the built-in profiling tools of the simulator.

Since the code itself serves as the model, the accuracy of the model is only dependent on how well the peripheral hardware is modeled. The hardware needs to correctly modeled both in terms of timing behavior and functionality, i.e. it must produce output data which is correct. When updating the application code, the model automatically gets updated. A lot of work can initially be saved by using this model, because the application itself does not need to be modeled, only the peripheral hardware.

While it is possible to run the actual code in an instruction set simulator it is important to note that the code is heavily optimized before it is deployed on the target hardware, and that it also is compiled without debug information. If the results from the instruction set simulator are to be comparable to the ones from the target system the same optimization and debug settings must be used when compiling the code. These factors disable the use of the instruction set simulators ability to halt execution at any point, check the state of any variable etc.

The most significant drawback of the model based on running the target code is the difficulty involved in finding relevant test data, i.e. worst-case test data, to feed the model. The test data also needs to be authentic, i.e. of the same format as data fed to the actual system. Finding the worst-case is based on running several tests with different test data

and then analyzing the profiling output from the simulator. However, the test data that provokes the system the most is not necessarily the actual worst case, only the worst case seen so far.

The instruction set simulator must emulate all the hardware present in the actual system in software, which makes it run slow compared to the real system. The slow execution speed can be mitigated by using a special add-on card for hardware acceleration. Also, the process of creating test data for the target system is non-trivial. Hence, the process of creating new test data (suspected to provoke the system more than previous test data) and then running the simulation can be quite tedious.

When prototyping new features, the level of depth in the model can be a problem. A change in one part of the code can affect other parts which might need adaption for the initial change made to the code. If the input data format is changed, i.e. to model a new feature, there is a great risk that the amount of work needed to change the model is up to par with doing the actual implementation. Thus, one is no longer prototyping the new feature, one is actually implementing it.

3.2.2 Abstract model

The abstract model is an abstraction of the actual system, running in a simulation environment. A complete abstract model needs to include abstractions of the application, the operating system and the hardware. There are simulation environments available which already model (real-time) operating systems. There is actually a simulation and analysis tool, VirtualTime [16], which contains a model of the real-time operating system used in the particular system analyzed in this thesis. Thus, by using this simulation environment the model engine itself would be ready, meaning that only the modeling of the application and the hardware needs to be performed.

With VirtualTime, one can create a model of the a real time system by using a C-library with functions for creating processes, inter-process communication, interrupts, etc.

A model of a system in VirtualTime can be seen as a set of processes that only contain the information needed for simulating CPU usage, delay characteristics and interaction with other processes. The resource utilization for system calls, context switches etc. can be customized. An example of VirtualTime code is shown in figure 3.1.

Since the model is detached from the real system it is relatively easy to analyze the impact of new functionality. The functionality only needs to be modeled in terms of CPU-utilization and delay has to be put in the right place(s).

Also, since the model only contains the parts important for the analysis, it makes it easier to understand the behavior of the system in different scenarios. Another benefit of this model is that, due to its relatively simplistic structure, it will have low execution times compared to the model running the actual target code in an instruction set simulator.

The abstraction of the system does not need realistic test data, in the sense that its input does not consist of a large byte array. It only needs the parameters necessary to accurately model the resource consumption when processing data. Thus, when prototyping changes in the input data, there is no need for tedious generation of accurate input data and rewriting other parts of the system to allow the new data format to pass through them.

The most significant drawback of creating an abstraction of the actual system is the uncertainty of how much initial work is needed to create such an abstraction, meaning that it is uncertain whether it is possible to create an accurate enough abstraction in the amount of time set aside for the implementation (around ten weeks). The abstract model also needs to be updated as the studied system evolves. When the system is updated, the new areas need to be profiled and the results must be integrated into the model.

Creating a model of the real-time system partly consists of analyzing portions of the code base to identify which parts to model. It is assumed that the bulk of the analysis has to be done manually, since the studied system is fairly complex. This assumption is supported by Axling in his master thesis [6] and also by Kraft et al in [10]. However, conducting a manual analysis should be feasible since it is not necessary to model all of the

```c
#include <vt_ose.h>
#include <stdio.h>
vt_process_t *process_ping;
vt_process_t *process_pong;

void ping_code(vt_process_t *me){
   vt_signal_t sig_snd, sig_rcv;
   for(;;){
      fprintf(stderr, "ping... "); fflush(stderr);
      vt_use_cycles(33);
      vt_send(&sig_snd, process_pong);
      vt_receive(&sig_rcv);
   }
}
void pong_code(vt_process_t *me){
   vt_signal_t sig_snd, sig_rcv;
   for(;;){
      vt_receive(&sig_rcv);
     vt_use_cycles(17);
      fprintf(stderr, "pong!\n");
      vt_send(&sig_snd, process_ping);
   }
}
int main(void){
   vt_init_simulation();
   vt_cpu_t *cpu = vt_create_cpu("CPU");
   process_ping = vt_create_process(VT_PRI_PROC, "process_ping",
                                    &ping_code, 5, cpu);
   process_pong = vt_create_process(VT_PRI_PROC, "process_pong",
                                    &pong_code, 5, cpu);
   vt_run_simulation(150);
   vt_exit_simulation();
   return 0;
}
```

Figure 3.1: *A simple VirtualTime code example, ping-pong.*

system, only the parts that are the major consumers of CPU-cycles and shared resources or that introduces significant delays. To identify these parts, profiling data can be used. Such profiling data have already been produced for the studied system, using a logic analyzer and probes in the code.

After the system analysis, the significant parts of the system are modeled, the model of each part would only capture the components' CPU-utilization and the delay they introduce in different scenarios.

When the modeling of the significant parts are finished a few scenarios can be run in the model, comparing the results with the results from the target system when using the same setup. Analyzing the deviation from the real system enables tuning of the abstraction to more accurately model the real system.

All these steps were feasible to complete in the given time period. However, whether the model would be able to fulfil the accuracy demands placed upon it was considered uncertain.

If the creation of an accurate enough abstraction of the system should succeed, it would feature some highly desirable properties. For example, prototyping new functionality is made relatively easily when compared to the model which runs the actual target code.

3.3 Requirements on the model

In this section the requirements on the model and their potential of fulfillment will be described. The phrase "Limited cost" is used throughout this section and might be considered to be a bit vague. It means that it should be feasible during the amount of time set aside for this project to at least gain the knowledge necessary to determine whether it is feasible.

3.3.1 Information output

The model should provide information about latency and CPU usage between defined points in the application. Results should be reported as minimum, mean and maximum for each connected UE. If possible, the model should also help identifying the worst possible combination of input data for the system.

Two measurement intervals have been identified, i.e. two sets consisting of a start and a stop point for which the CPU- utilization and latency for each user data frame is studied. The latency is calculated from the point where the system obtains the user data to the point where the processed user data leaves the system. Depending on the user data, there are different requirements on the maximum time allowed to complete the processing of a certain type of user data block. There are no actual requirements on the level of CPU-utilization, but it is still of significant importance as it is important to track the CPU-utilization for each target code change.

Model based on target code

The instruction set simulator which supports the CPU used in the studied system supports execution of code in a scope external to actual simulation, such code does not consume clock cycles on the simulated CPU and can be triggered when writing to a certain area in memory. It may therefore be used to log data non-intrusively, essentially the same way as a logic analyzer is used when measuring on the actual system. This data may be saved to a file, which can be formatted as desired.

Information about the worst possible combination of input data is hard to derive because the model is nearly as complex as the system itself. Also, real input data, the creation of which is non-trivial, must be used. Input data will be discussed in section 3.3.4.

Abstract model

The simulator of choice supports the insertion of measurements points which can output information on both CPU-utilization and delays to a text file. This text file can then be converted to a desired format.

Information about worst possible combination of input data is relatively easy to find, as the model itself solely is built and based on how certain input data affects the system. Also, the input data is abstracted and the creation of new input data is trivial (see section 3.3.4).

3.3.2 Accuracy

To provide valuable results the model needs to be accurate. In this particular study the goal is to achieve a level of accuracy within ±1% for minimum/maximum Millions of Cycles Per Second (MCPS) (CPU usage) and ±3% for minimum/maximum latency.

The model based on target code has the best chances of becoming sufficiently accurate. However, in a case study done by Wall et. al [17] an abstract model of a system consisting of 60 tasks and over 2.5 million lines of code was created. Their final model consisted of six tasks and 200 lines of code and still provided valuable results. The system studied in this thesis is fairly small compared to theirs, both in terms of number of tasks and lines of code. It is therefore reasonable to argue that it is possible to create a complete abstraction of the system, both for hardware and software. While the amount of time that went into their work is unknown, it shows that it is possible to have a large level of abstraction and still getting valuable results.

Reaching the stated accuracy level with the abstract model will without doubt be a difficult task. In a worst case scenario, in order to reach sufficient accuracy for every possible input data combination, one ends up with a model as complex as the real system but the model does not actually do anything except calculate CPU-utilization and delay.

3.3.3 Verification

At every change in the actual system, the model must be updated and verified to make sure that it is reliable. When making minor changes in the product and implementing the corresponding changes in the model the *robustness* of the model may also be verified. Robustness implies that making a change in the actual system and the model should yield the same results regarding the modeled properties.

Model based on target code

To verify that the model is correct, one runs a test case through both the model and the actual system, if the results are coherent the model is correct. Should the results differ, the deviation adhere from the models hardware part or from the configuration, since the source code is common between the model and the actual system.

Abstract model

As in the model based on running the actual target code, a test case would be run in both the model and the actual system. However, deviations in the results could originate from any part of the abstract model. Thus, pinpointing of the actual error source might be harder compared to the model running the target code. The process of searching for the source of error is likely to result in findings of additional factors important for the timing properties and thus resulting in further refinements of the model.

3.3.4 Input

The model input largely differs between the two models. In the model based on target code, actual input data must be used. In the abstract model the input data can be abstracted to only contain the properties necessary for the simulation, i.e. abstracted to a set of parameters that affects the processing speed of the input data in the system. The abstract

model makes it possible to omit data which is of no interest when performing detailed studies.

Model based on target code

This model utilizes the same code base as the actual system, thus the format of the input data is the same. Although it takes a lot of work to generate this input data, it has already been done for some test cases when running profiling tests on the actual system. As of now, there exists around 10 - 15 test cases containing different types of user data representing different scenarios.

Besides user data, control data also has to be simulated in this model. The system must be set up correctly before it can begin processing user data, and since the software model corresponds to the actual software, this type of input data must also be 100% authentic.

Abstract model

The abstract model allows the input data to be largely abstracted, since no actual processing of the input data is done. A block of real input data would be modeled as a structure containing different parameters, where the parameters decides how the block affects the system, the type of the data and which user the data is bound to. Constructing test data is a lot easier when using this model, as it is not necessary to construct it on bit level detail. Also, the control data can be completely omitted in this model.

3.3.5 Output format

Making the output from the model correspond to the output format used when debugging the actual system has some benefits. Existing tools for analyzing the log files from the measurement on the actual system can be used on the output from the model. This makes comparison of the model and the actual system easier.

Model based on target code

The probes in the actual system can be re-used in this model. The output format is hence the same, and there is virtually no work involved in matching the model output to the systems output.

Abstract model

To match the output from the actual system, the abstract model would need to be fine grained enough to include all measurement points from the actual system. This could be done by using the existing measurements points as corner stones when creating the model. Making the output format correspond to the actual system would be a trivial task, since it is a matter of simple formatting.

3.3.6 Limited cost of modeling current software

If there is too much work involved in creating the model of the current application software, its value will not exceed the costs for creating it.

Model based on target code

The cost of modeling the current application software is zero. The code itself serves as the model, and therefore virtually no work is needed.

Abstract model

All processes and interrupt routines must be created to match the actual system. The interaction between processes must be investigated by looking at the target code. In each process, the level of abstraction must be decided. Profiling data must be obtained in order to simulate CPU-utilization and delay of the different parts of each process. By manual investigation of the code in conjunction with analyzing profiling data of different inputs,

it should be possible to derive parameters from the input test data that influences how different blocks or functions in the code scales in terms of CPU utilization and delay. The work needed to complete this process is largely dependent on the level of abstraction needed in each process.

The real-time operating system must also be modeled. For the purposes of this thesis this is taken care of by VirtualTime. The size and complexity of the studied system put a limit on the degree of simplification that can be achieved without loosing too much accuracy.

3.3.7 Limited cost of modeling current hardware

Besides the application software there are peripheral hardware components which need to be modeled. As with the application software, the amount of work needed for the process of modeling the hardware has to be reasonable.

Model based on target code

Experiments with modeling some of the peripheral hardware had previously been performed at the company developing the studied system. The modeled hardware may be integrated with the application code and compiled into a binary file, which can then be run in the instruction set simulator. This can probably be re-used in the project to some extent. However, a general problem of modeling hardware for use with the model based on running the actual target code is the detail level needed, since the software expects the hardware to use a certain input and output. Fortunately, the simulated hardware does not need to process the actual user data. Only correct length of the output data is required as the user data more or less leaves the system after passing through the hardware, i.e. no further processing of the user data takes place. In the EDCH case the data will be further processed after TDP processing. This increases the complexity of the input data generation, described in section 3.3.4.

Abstract model

In the abstract model, only the delay introduced by the hardware needs to be modeled. The delay is probably dependent on the different parameters of the input data. Some hardware, e.g. the DMA controller, might be more complex to model than other. The hardware which is to be modeled must be profiled in the same manner as the target code.

3.3.8 Limited cost of modeling changes in software

One of the potential usages of the model is the ability to prototype new features in the software and see how they affect the system.

Model based on target code

In the model based on running the actual target code, the approach for modeling changes depends on the state of the change to be modeled, i.e. if the change is already implemented or is to be prototyped.

- Before implementation

  As long as only addition of code is necessary, i.e. no code removal, only CPU-utilization and actual delay need to be modeled. If however code removal is necessary, this could be problematic since the removal of code can have impact on other parts of the code, and also on the input data being processed, which can result in invalid in-data for different processing blocks in the code. Another problem in this model is the input data. If the feature to be modeled requires a new format of the input data this can be problematic. The creation of real input data is a non-trivial task, and the creation of real input data which has a new format is even harder. Also, the whole model needs to be adapted to the new format.

- After implementation

If the change is already implemented, the model is automatically updated since the application code itself serves as the software part of the model.

Abstract model

In the abstract model, only CPU utilization and actual delay need to be modeled. The process is similar regardless of whether the model is updated before or after the actual implementation of the new feature. It might be necessary to refine some parts of the model to accurately be able to insert the new resource utilization. If so, this can be done by doing refine measurements in the actual system of the involved parts and update according to the gained results.

3.3.9 Limited cost of modeling changes in hardware

Fulfilling the requirement "limited cost of modeling changes in hardware" gives the benefit of being able to test different peripheral hardware configurations to see how they affect the system.

Model based on target code

Some hardware, such as the CPU itself, the DMA controller and the TDP, are included in an existent instruction set simulator for the CPU studied in this thesis. This hardware can not be changed, so changes in that hardware must be modeled in the software, i.e. by wrapping the hardware. This is hard and makes the boundary between hardware and software less clear. Changes in the modeled hardware should be of equal complexity as the initial modeling of it and involves the same difficulties, i.e. the need for detail since the software expects the hardware to behave in a specific way.

Abstract model

In the abstract model, the process of modeling changes should be equally complex for all types of hardware since all hardware are an abstraction. Modeling changes in the hardware is quite simple as the dependencies and interaction between the software and the hardware can be decided.

3.4 Use cases

In the following section three concrete usages of the model will be looked at, i.e. typical applications of the model which motivates its creation. Each usage can be seen as one or more of the requirements somewhat concretized. For each usage, a method of realization in each model will be described.

3.4.1 Modeling new application features in early project phase

The possibility of detecting the impact of a new feature in an early project phase represents one of the main usages of the model. As stated in chapter 1, this is the main motivation for this thesis.

Model based on target code

When prototyping new features in an early project phase for the model based on running the actual target code, the resource utilization in terms of delay and CPU time of the new feature must be estimated. This requires the engineers implementing the new feature to make a reasonable guess. Second, the placement of the resource utilization in the target code must be found. A benefit of this model is the accuracy of placement because of the absence of abstraction in the software model. The new feature may only be triggered when processing a specific type of user data, i.e. a conditional execution. This type of user data might not even exist yet, as the feature itself is only at the prototyping stage. And even

if it were to exist, it could have undesirable impact on the other processing steps as the format has changed. The conditional execution must hence be implemented by utilizing breakpoints in the Instruction Set Simulator (ISS) to trigger execution of code looking at data outside the application, simulating new user data. When testing the new feature, the new test case would be based on an older test case and looking outside the application for parameters deciding the conditional execution.

Abstract model

For the abstract model, the first steps of prototyping new features in an early project phase are equal to the target code based model, estimate resource utilization and placement in the code. However, refinement of the model might be necessary if the abstraction level of it is too high. Conditional execution is more straight forward in the abstract model, since new parameters can be added to the input data with ease without affecting other blocks. The new parameters are simply ignored, except in the block modeling the new feature. When testing the new feature, the new test case would be based on an older test case but with the addition of a new parameter simulating the new format of the user data.

3.4.2 Identifying worst case

The possibility of identifying the worst case has obvious benefits. Without a probable worst case, the measurements performed on either the real system or the model are of limited use, since they only would predict how the system would act during usage less intense compared to the actual worst case. This means that the system could fail to meet its deadlines when being exposed to the actual worst case.

Model based on target code

Constructing test data to represent the worst case scenario for the model based on running the actual target code is hard, not only because of the work involved in constructing

input data, but also because it is hard to actually identify the worst case. The process of identifying the worst case would consist of running several test cases with different input data and see which test case made the worst case. However, this does not mean that this was the actual worst case, only the observed worst case. To later analyze what made the specific test case being the observed worst case is a non-trivial task, since the software model is highly complex because of the absence of any abstraction.

Abstract model

Identifying the worst case in the abstract model is predicted to be easier because of two major reasons. The first reason is that the creation of test data is much simpler since it is an abstraction. More test data allows more testing and better chances of finding the worst case. The second reason is that the abstraction of the system, which yields in much lower complexity, allows easier understanding on how different types and amount of user data affects the system. Hence, the model can be examined and give indications of what a worst case test case should look like.

3.4.3 Identify bottlenecks in the system

Identifying bottlenecks in the system means trying to find areas in the application to improve, i.e. sections which have significant impact on the performance of the system. For example, one might be interested in identifying resources that greatly influences the execution speed of some test case. Resources which are at the limit of their capacity are also interesting, since minor changes of the input data may result in substantial effects on the execution speed.

Model based on target code

The instruction set simulator for the CPU the studied system runs on has profiling tools available, so measurement points at the start and end of the block of interest may be

placed in the code. The placement can, as stated previously, be very accurate as there is no abstraction when the code itself serves as the model. The simulation is then started, and when finished the results from the profiling can be analyzed.

Abstract model

The approach in the abstract model is similar to the target code based model. Measurement points are placed into the code. However, if the abstraction level at the wanted location of the measurement is too high, refinement of the model is needed. For example, if a measurement is to be performed inside a block in the model, the block needs to be split into smaller parts. Hence, a higher level of detail and lower level of abstraction is needed. The block is split up, and detailed profiling is performed, using either the ISS or the target system. This produces profiling results which are imported to the abstract model.

3.5 Prioritized requirements

After having conducted the feasibility study, the work required to create any of the two models where found to be substantial. Both models clearly have both benefits and drawbacks, and in order to make a choice the most important requirements and their predicted possibility of fulfillment in each model where identified. The most important requirements where identified as being accuracy, finding the worst case and to model changes in the software (either new features or new designs). The hardware is not expected to change as often as the software, so this is of less importance.

3.5.1 Accuracy

It was believed that the model based on target code would reach the required level of accuracy. In contrast, it was considered uncertain if the abstract model could reach sufficient accuracy in the amount of time set aside for the project. However, it was considered pos-

sible, and that it would also be of interest to see what level of accuracy could be reached. It would also be of interest to see which parts needed the highest detail when modeling. Also the model creation process itself could give many valuable lessons.

3.5.2 Identifying worst case

The difficulty of identifying the worst case in the model based on target code is of the same magnitude as identifying it by measuring on the actual system. In the abstract model, it is easy to create new test cases, and it also helps identifying the worst case by offering lower complexity which gives easier understanding on how input data affects the system.

3.5.3 Model changes in the software

Smaller changes, i.e. increased processing time of a certain block should be of equal difficulty in both models. However, since the software model in the model based on target code is the actual software, the modeling of larger changes, e.g. involving new input data, changes in the design or architecture, becomes more difficult. It is possible that the modeling starts to become the actual implementation in order to get the model to work. The abstract model, because of its important property of actually being a model of the software, better lends itself for this type of changes.

3.6 Model choice

It was decided to go with the abstract model, since it has a number of benefits compared to measuring on the actual system and the model based on target code. The uncertainty of the accuracy level of the abstract model is a risk, but it was decided that the benefits made it a risk worth taking. Also, should the model prove to be too inaccurate, it might be interesting to see which parts that need refinement in order to make it accurate enough. Even if the model fails to be accurate enough when this thesis is finished, it has probably

given important information about the difficulties when creating such a model and also information about the system itself. By trying to create an abstraction of the system, knowledge of the system is needed and it must be closely examined. This process can lead to insight of the system.

The model creation process was expected to provide insight into some key questions, in the event that creation of the model were to fail, for example:

- Which sections forms the complex parts of the studied system (are hard to model)?
- What makes a section too complex to be accurately modeled?
- Which level of accuracy can actually be reached?
- What would it take (more work, time, tools etc.) to be able to make an accurate abstract model of the system?

3.7 Summary

In this chapter two different approaches for creating a model, a model based on target code and an abstract model, were presented. The feasibility study, which was the process of identifying requirements on the model and how the two different approaches could fulfill these requirements, was also described. Finally, the most important requirements were identified and the choice of the most appropriate approach, which turned out to be the abstract model, was made.

Chapter 4

Model creation methodology

4.1 Introduction

This chapter will describe the process and methodology of creating an abstract model of the studied real-time system. Two different sources of measurement input for the model will be presented, namely an instruction set simulator and event logs from the target system. Finally, an introduction to VirtualTime and the methodology used for creating models utilizing it will be given.

4.2 Methodology overview

The first goal of the modeling process was to identify the different parts of the system. This was done by looking at various technical documents, obtaining information from engineers working on the system and also by examining event log files from the system created during testing, i.e. during execution of the system on real hardware using real input data. When the different parts had been identified (presented in section 2.4.1), the goal was to find out how these parts interacted with each other. This was accomplished in a similar manner. The next step, which also turned out to involve most of the practical work, was to find how

the different parts of the system consumed the two studied resources, namely CPU-cycles and time. It was previously known that different parts of the system consumed different amount of cycles and resulted in a different amount of delay depending on the input data currently flowing through a specific part of the system. The goal was hence to derive how different types of data influences the resource utilization in the different parts.

4.3 Input sources for the model creation process

When creating the model of the system two main types of input were used, manual code analysis and profiling results.

First, the actual C code that the system is composed of was manually analyzed looking for parts which were believed to use a variable amount of clock cycles to execute depending on the input data. In the second step the code was run in an instruction set simulator, thus providing cycle accurate information on the actual execution times for the tasks identified in the manual code analysis. When running the application, certainty that no important parts were missed in the manual code inspection step could be reached. By doing this type of inspection combined with measurements, it was considered possible to derive formulas for calculating the cost of different functions and see how the cycle consumption depended on different parameters in the user data.

There were only four different types of input data available for the system when running in the ISS. Using input data from the actual system was not possible, since the code running in the instruction set simulator works slightly different. This was partly because of the lack of some hardware components and because running the system this way is currently a work in progress. There was also a need for verification of the measurements and the formulas derived from running test cases in the ISS. Therefore, event logs from the target system were also used. The event logs contain information that were output from the system during execution on real hardware. Hence, if the event log says it takes a ceratin

amount of time between point A and B in the program, this is really the time it takes.

The reason for not solely relying on the event logs from the target system as input for the model is that the possibility of doing measurements in the target is limited to log time stamps with some small additional info, e.g. a location in the code paired with the value of a parameter. There is also a probe effect which can be significantly interfering when measuring very small pieces in the code, e.g. an iteration in a loop. Also, the procedure of running a target test on real hardware involves high effort compared to just loading the system into the ISS. When moving measurement points, the code has to be recompiled, delivered to the test department, test nodes must be booked, etc. As long as the measurements made in the ISS can be verified, the ISS is a great help both when trying to identify good points for measurement and also for doing the actual measurement.

The method of obtaining profiling results from the instruction set simulator and verifying these results against the target system will now be described.

4.3.1 The instruction set simulator used for profiling the code

An instruction set simulator was used to ease the analysis of the system. The instruction set simulator utilized contains a cycle accurate model of the CPU, the TDP, memory and the DMA controller which the actual system runs on.

The ISS used supports profiling of functions, loops and ranges of code to see how many clock cycles are consumed by a particular section and how many accesses that are made to them. It is also possible to insert breakpoints in the code, halting the execution when control reaches that point. Breakpoints may be used in conjunction with single stepping through the code to get an accurate view of the application's flow.

A new, limited, environment for running profiling test cases in an instruction set simulator was deployed, Test Bench (TB). Four different test cases were provided for evaluation of the decoder. Since the test bench part is integrated in the regular system code, it is not possible to use the TB for doing performance and capacity tests, as the test bench part

executes on the same CPU and hence, at least in theory, interferes with the system. The possible interference caused by the TB when doing measurements will be discussed further in chapter 5.2.4. The goal of the TB is to test functionality on small amounts of user data. When referring to doing measurements etc. in the ISS, it is the TB loaded into the ISS that is referred to.

As mentioned earlier, there are four different test cases available for the TB. The data which forms each test case has a unique Enhanced Transport Format Control Indicator (ETFCI). The user data in the abstract model consists of a set of parameters that characterize the user data, in terms of CPU cycles consumed and latency. The most important of these parameters was found to be the ETFCI. The ETFCI decides the value of a number of different parameters, i.e. the properties of the user data.

4.3.2 The target event logs

For verification of the measurements in the ISS, event log files which were output from the actual system during stress tests were used as a source of input for the analysis. The logs are created by inserting code snippets in the target code that writes information about the current event or location in the code, the current executing process, a time stamp and additional info depending on the type of event. This information is written to a certain address range in the external memory which is monitored by a logic analyzer. As soon as there is a write to the address range, the logic analyzer triggers and dumps the information into a log file.

4.4 VirtualTime

An existent tool was chosen for creating the abstract model, namely VirtualTime. This tool was chosen since it contains a model of the real-time operating system used in the studied system. VirtualTime is available for Linux, Windows and Solaris. The Linux

version was chosen because it was available for the latest version of VirtualTime.

According to the marketing material at the vendors site [16]:

> “VirtualTime is a tool-set for building accurate simulations of complex multi-processor real-time systems.”

VirtualTime is created by Rapita Systems [12] in collaboration with ENEA[7], which is the vendor of the real-time operating system OSEck[11]. VirtualTime contains an abstract model of the operating system OSEck, and thus it is a suitable tool to model systems running on top of that operating system.

When modeling in VirtualTime, it is fairly easy to capture the real-time systems flow of control, since the library contains calls which correspond to the calls in OSEck. For example, to send a signal to some process in VirtualTime, *vt_send(vt_signal_t*, vt_process_t*)* is used and the corresponding method in OSEck is *send(union SIGNAL **sig, PROCESS to)*. The similarity of the VirtualTime library and the OSEck system calls makes it fairly straight forward to map the implementation to the model, and makes the model easy to understand for programmers who are familiar with OSEck.

4.4.1 VirtualTime entities

VirtualTime provides a number of data types and methods which are used when creating a model, some of which are seen in the ping-pong example code (figure 3.1). The data types and methods used in the ping-pong example are described below along with additional important entities.

vt_cpu_t

The type *vt_cpu_t* is used to identify the virtual CPU s in VirtualTime models. Variables of this type are assigned a value returned from the method *vt_create_cpu()*. The variable is

then used when registering processes with *vt_create_process()* or *vt_create_external_process()*, in order to bind them to a specific CPU.

vt_process_t

vt_process_t is the data type used to identify processes in VirtualTime. The intended use of variables of this type is to assign them a value returned by *vt_create_process()* or *vt_create_external_process()*. These processes may be written much like the processes in the modeled system, except that the only factors that needs to be modeled are resource usage and timing in the case of non-external processes. When modeling external processes only the timing behavior needs to be modeled. One may also opt to merge several processes into one in the model if one wishes to view these as a big block of resource usage.

vt_create_process() and vt_create_external_process()

The calls *vt_create_process()* and *vt_create_external_process()* are used to register processes and external processes respectively. As might be expected ordinary processes are bound to a CPU, to be able to get the correct timing and consume clock cycles. External processes are also bound to a CPU, in order to get correct synchronization information.

vt_signal_t

vt_signal_t is used when sending signals between processes. This type is defined as a C-struct with, among others, the following members *vt_uint32_t sig_no*, *void *mem*. These fields represent the signal number and a pointer to a memory block associated with the signal.

vt_use_cycles() and vt_delay_until_cycles()

The method *vt_use_cycles()* is used to consume cycles (and hence also time) on a virtual CPU. The cycles are automatically tied to the calling process and thereby the CPU-

utilization of different parts may later be analyzed.

The method *vt_delay_until_cycles()* is used to cause a process to delay its execution until the virtual clock of the CPU it is bound to reaches the value specified in the function argument. Unlike *vt_use_cycles()* this call consumes no clock cycles.

vt_send(), vt_receive() and vt_sel_receive()

vt_send(), *vt_receive()* and *vt_sel_receive()* are the methods used by processes to send and receive signals. If a process is only to receive a signal of some specific type(s), the type *vt_sig_select_t* is used in conjunction with the method *vt_sel_receive()*. An example of how to use *vt_sel_receive()* is given in figure 4.1. *vt_receive()* and *vt_sel_receive()* are blocking calls, meaning the calling process will not continue to execute until it receives a signal.

vt_init_simulation()

vt_init_simulation() is the call used to initialize the simulation library and get a license key for VirtualTime. This call must be made prior to calling any other VirtualTime Application Programming Interface (API) calls, except the calls to set the error handler and log file.

vt_run_simulation()

vt_run_simulation() is used to start the actual simulation, it takes one argument, *sim_length*, of type *vt_time_t* which is the length of the simulation in *vt_time_t* units. The function returns when a CPU's virtual clock reaches the time *sim_length*.

```
/* Code snippet illustrating how to only accept a signal of some
   specific type and act differently depending on the type of
   signal received. */
vt_signal_t s;

/* 2: the number of signal types to accept,
   IO_DATA_AVAILABLE, IO_DATA_PROCESSED: the signals to accept */
vt_sig_select_t sig_list[3] =
    {2, IO_DATA_AVAILABLE, IO_DATA_PROCESSING_READY};
for(;;){
    vt_sel_receive(&s, sig_list);

    switch(s.sig_no){
        case IO_DATA_AVAILABLE:
            handleIODataAvail(); break;
        case IO_DATA_PROCESSING_READY:
            writeData(); break;
    }
}
```

Figure 4.1: *A code snippet illustrating how to receive a signal of some specific type(s) in VirtualTime.*

4.5 Use of VirtualTime in the model

The VirtualTime entities detailed in section 4.4.1 were utilized when creating the model of the system. The information provided in this section is an overview, for detailed information about the implementation in VirtualTime, please refer to the bundled source code.

4.5.1 Modeling of processes

When modeling the processes in the system, regular C-functions was created containing the functionality needed to model cost and flow of execution. When starting the simulation, these processes were registered with the simulator by using the call *vt_create_process()*. See the ping-pong example in figure 3.1 for an example of this.

4.5.2 Modeling of hardware

A hardware resource can be viewed as a process which provides some service with some arbitrary delay, but without consuming clock cycles on the CPU. Although external processes do not consume any clock cycles they are bound to a CPU, for timing purposes. When modeling hardware resources the code was bound to an external process by utilizing the call *vt_create_external_process()*.

4.5.3 Modeling of software functions

When modeling functions, it was often opted to simplify them into functions returning the clock cycles which would be consumed by the functions in the actual system. An example of a simple cost calculation function is given in figure 4.2.

```
/* A simple example of a function returning the amount
   of clock cycles consumed by bubble sort.
   n: Number of elements.
   k: Coefficient.
   m: Point of intersection. */
vt_cycles_t bubble_sort(unsigned int n,
                        unsigned int k,
                        unsigned int m){
   return (vt_cycles_t) k*n*n + m;
}
```

Figure 4.2: *A code snippet returning the amount of cycles consumed by a bubble sort implementation.*

4.6 Model limitations

4.6.1 Only user data of type EDCH

The decoder can handle different types of user data which can be divided in two main groups: Dedicated Channel (DCH) and EDCH. DCH is mainly used for speech and lower data rates while EDCH is used for high speed data transfers. The process of handling the user data is different between the two and involves the use of different components, both software and hardware. In this thesis the study is limited to the EDCH part of the system. Because of the limited time available, it was decided to focus on one part only and getting it as accurate as possible. The results and knowledge gained can then be used when modeling the DCH part of the system, or maybe help coming to the conclusion that it is not possible or feasible to create an abstract model of the decoder. Everything explained in this section, unless explicitly stated otherwise, hence concerns EDCH and might not apply for DCH.

4.6.2 Only user data with 2 ms TTI

For EDCH, there are mainly two different types of user data, 2 ms TTI and >=10 ms TTI. Only 2 ms TTI have been considered in this model.

4.6.3 Omitted control plane

The control plane, i.e. the sending and receiving of signals used to configuring the decoder has been completely omitted. The control plane is believed to have very small impact on the overall performance as it is used mainly for startup, adding and removal of users and reporting results from various measurements.

4.6.4 Modeling limited to two components

Because of the limited amount of time available, there was only time to thoroughly analyze and base the model on two of the components in the system: The application scheduler and the TDP. Because of this limitation, much hardware interaction and hence real-time effects are also omitted.

4.6.5 Omitted retransmissions

If the decoding of a sub frame fails, the decoder can save the sub frame and order a retransmission of that particular frame. Upon receival of the new sub frame, the decoder can make use of both sub frames when doing the decoding. This increases the chances of a successful decoding as the two versions of the sub frame can compensate for each others bad parts. This behavior is omitted in the model and all sub frames are presumed for successful decoding.

4.6.6 Little conditional execution

All user data takes an almost identical path through the model, e.g. no retransmissions as explained in section 4.6.5. There are places in the target code where different paths might be chosen because of different attributes of the current user data. This needs to be modeled further by either refinement of the abstracted user data to contain these parameters and having conditional execution in the model or by using some sort of random selection of cost.

4.6.7 Only one user

As not all components are modeled, the model is limited to only one user. This is because the interaction of having many concurrent users in the system requires all components to be accurately modeled, in order to show the real-time effects of having many users affecting each other.

4.6.8 Summary of limitations

The limitations are, in short: the model supports only traffic of type EDCH with 2 ms TTI, it does not support retransmission of data, it does not support more than one user, the control plane has been omitted, many special error handling parts have been omitted and it only models two components of the studied system.

Although the limitations are substantial, many types of normal EDCH traffic types are still covered.

4.7 Modeling the different parts

In this section the modeling process of all parts will be described. The exact accuracy and results from the modeling will not be discussed in this chapter, they are instead placed

in chapter 5. Due to time constraints, only the application scheduler and the TDP have been closely examined and modeled. The other components are modeled only so that the flow of events is correct, i.e. the correct sequential interaction between the components for handling a piece of user data. These components will be discussed in terms of thoughts about what the process of complete modeling of them would look like.

4.7.1 Application scheduler

The application scheduler is responsible for most of the user data processing, and also the frame protocol building, i.e. the packaging of the user data before it leaves the system. This is where most of the cycle consumption takes place. It can roughly be said to contain a loop where a signal is fetched from a signal queue each iteration. Depending on the signal, different actions are taken. This rough picture of the application scheduler is the abstraction level chosen for the model. In reality there are many types of signals being sent to the application scheduler, but since the control plane is completely omitted, solely the two signals involved in the processing of 2ms TTI EDCH user data were studied in this thesis:

- *DATA_FB_READY*

  This is the signal sent to the application scheduler from the DMA interrupt process when the user data has been copied to the internal memory and is ready for processing.

- *DATA_TDP_READY*

  This is the signal sent to the application scheduler from the DMA interrupt process when the user data has been processed by the TDP and is ready for frame protocol building.

For each signal, there is a function that handles the corresponding signal. The goal was to find how the amount of cycles used for each function was influenced by different types

of user data. The method for obtaining this information was to run the system in the ISS and stepping through the functions. While stepping, loops were examined while keeping an eye on the simulator clock, which shows the amount of cycles used, to see where most of the cycle consumption originated from. As expected, most of the cycle consumption came from loops iterating numerous times. By calculating the cost for one iteration (which often was constant) and looking at the variable(s) deciding the number of iterations, it was possible to derive formulas for calculating the cycles consumed by such loops. The difficulty of creating such formulas differed between different loops. Sometimes it was necessary to copy some parts of the original code because of non-trivial dependencies between variables, loop cost and the number of iterations. By identifying a parameter in the user data that influenced the amount of cycles consumed, it was possible, by doing measurements with different user data, to derive a formula. The parameters bound to the user data that was found to affect the cycle consumption would be the contents of our abstracted user data. Where applicable, linear mathematical formulas were derived from the measurements by utilizing the least-squares algorithm.

For the application scheduler, six functions whose cycle consumption varied significantly for the different types of user data were identified. The remaining costs were mostly static but with a slight increase for more dense user data. The modeling of each six functions and the remaining costs will now be described.

doScaleAndQuantization()

The cycle consumption of this function depends on the parameter *SymbolType* (shown in table 4.1) and the number of symbols. Each *SymbolType* has a different cost per symbol. For the available test cases, the user data only had two different values of the *SymbolType*, for which the cost per symbol was examined. There was also some additional costs found to be linear to the number of symbols on each physical channel regardless of the *SymbolType*.

SymbolType	Cycle cost per symbol
1	1.5948
2	2.8583

Table 4.1: *The amount of cycles used per symbol for different values of the SymbolType parameter*

do2ndDeInterleaving()

The clock cycles consumed by the second deinterleaving function, *do2ndDeInterleaving()* was found to scale linearly with the number of symbols of the data being processed. The relationship between cycles consumed and number of symbols is illustrated in figure 4.3.

doRateDeMatch()

When a data channel with a certain bandwidth is requested by a user the characteristics of the actual data channel acquired is selected from one of two tables describing transfer modes. Even though there are two tables, one logarithmic and one linear with 128 and 126 entries respectively [3], the probability of getting a channel which maps exactly to the requested bandwidth is low. This creates a need for the user equipment to perform *rate matching*, which means modifying the bitstream by removing or adding bits so that it fits exactly into the supplied transport channel.

Once the bitstream reaches the decoder it needs to be rate *de*matched. When performing rate dematching the decoder needs to traverse the bitstream, changing the value of punctured/repeated bits as it goes along. See figure 4.4 for a graph showing the correlation between $x = [numberOfCodeBlocks] * ([codeBlockSize] + 4)$ and the amount of cycles, y, consumed by the function *doRateDeMatch()*.

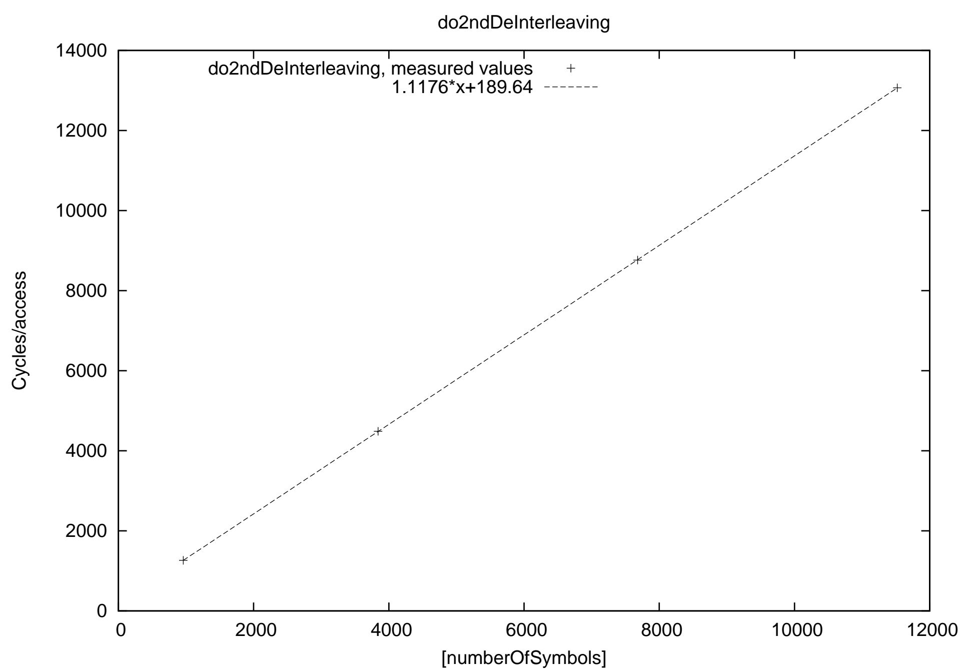

Figure 4.3: *A graph illustrating the correlation between y = the cycles consumed by the second deinterleaving step and x = the length of the data being processed.*

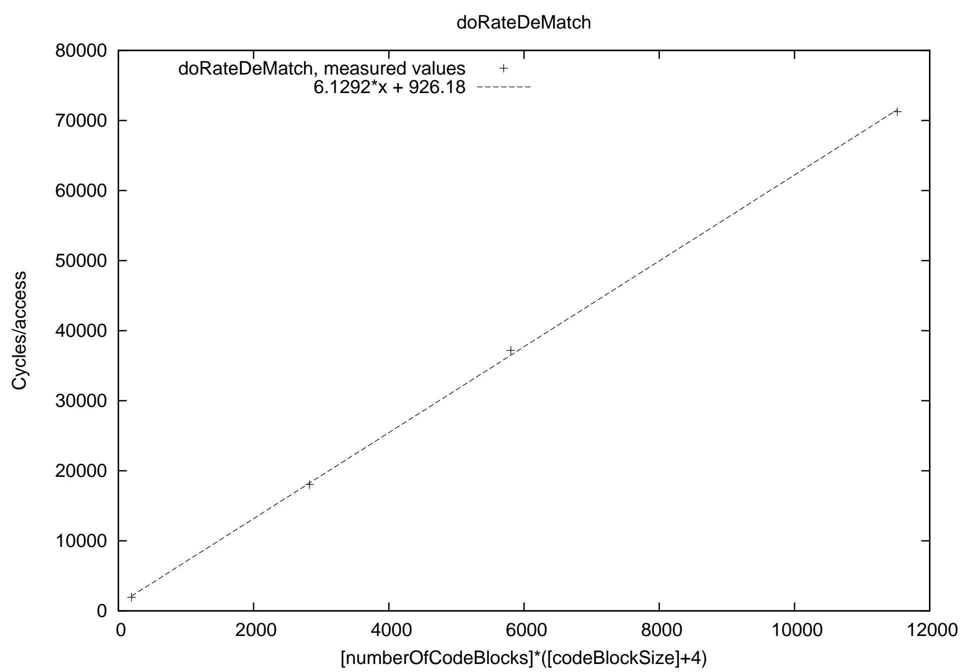

Figure 4.4: *A graph illustrating the correlation between* $x = [numberOfCodeBlocks] * ([codeBlockSize] + 4)$ *and* y *=the amount of cycles consumed by the rate dematching step.*

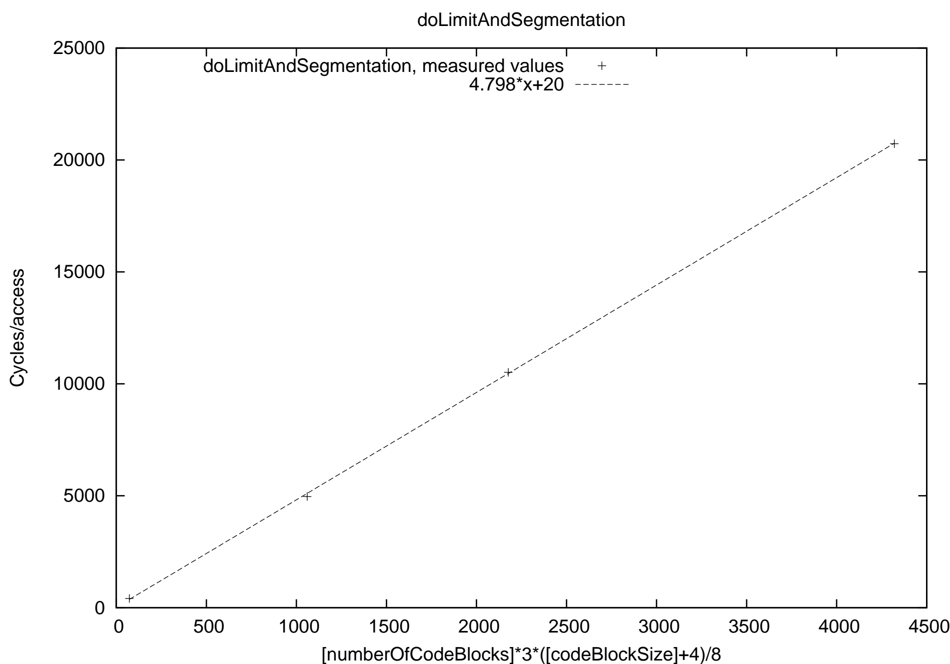

Figure 4.5: *A graph illustrating the correlation between the cycles consumed by the limit and segment step (y) and a mathematical function composed of the number of code blocks and the code block size (x)*

doLimitAndSegmentation()

The function *doLimitAndSegmentation()* depends on the number of code blocks and the code block size.

Most of the costs came from a loop with a static iteration cost. The number of iterations could be seen when inspecting the code and is calculated with the formula $\frac{3}{8} * [numberOfCodeBlocks] * ([codeBlockSize] + 4)$. By using the measured cycles as the Y-value, and the results from the formula stated above as the X-value, a linear equation of the form $y = kx + m$ could be derived, see figure 4.5.

calcTdpParams()

The function *calcTdpParams()* depends on the code block size and the number of code blocks.

While the cycle consumption of some sub portions of the function were easily calculated, there was one particular sub function, *fetchInterleaveTable()*, which cycle consumption was not easily calculated at first. After closer inspection the cause of this was suspected to be an ongoing DMA job coupled with a loop where the CPU read from the external memory in each iteration, while the DMA job was still being processed. This would increase latency of fetching the data from the external memory, as these accesses also uses the DMA. This theory was verified by placing a code block consisting of a loop prior to the previously mentioned one, in order to let the DMA job complete. When letting the DMA job complete prior to entering the loop where the CPU read from external memory, the loop consumed a constant amount of cycles per iteration. The ISS showed that the reduction of cycles originating from waiting for the DMA controller to fetch data into cache constituted the entire decrease of consumed cycles, additional proof that the root cause of the phenomenon had been identified.

The finding about conflicting DMA jobs is an important observation, since it illustrates that the execution speed of code blocks utilizing external memory may vary by considerate amounts depending on the state of the DMA controller. This finding will be further discussed in sections 5.2.4 and 6.4.4.

Because each test case is run during identical conditions in the ISS, it was possible to derive a formula since the latency introduced by the external memory accesses were predictable. This is due to the low DMA load when only running one user. This formula will most likely not hold when the system load increases, because the DMA will then behave more unpredictably.

The observed behavior of the function *calcTdpParams()* is depicted in figure 4.6.

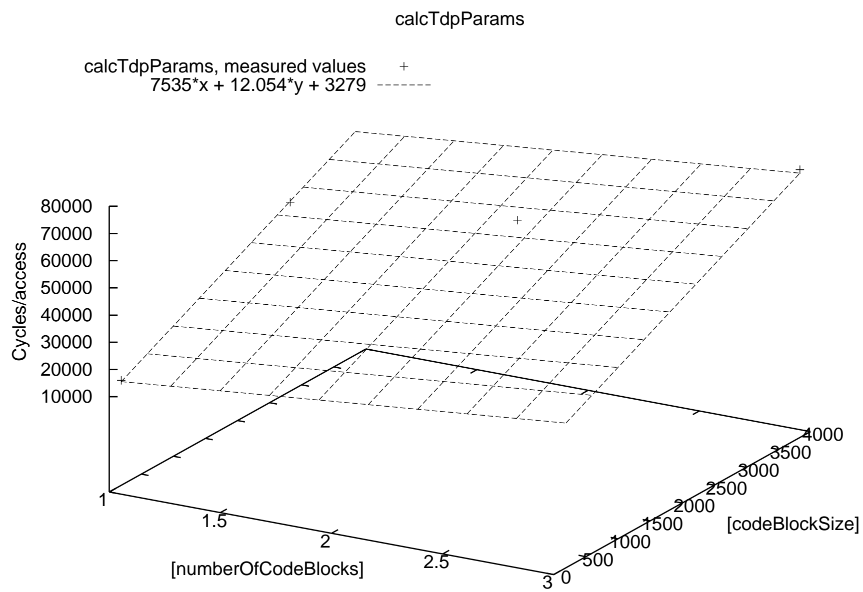

Figure 4.6: *A graph illustrating the correlation between the cycles consumed by the function calcTdpParams (z) and a mathematical function composed of the number of code blocks (x) and the code block size (y)*

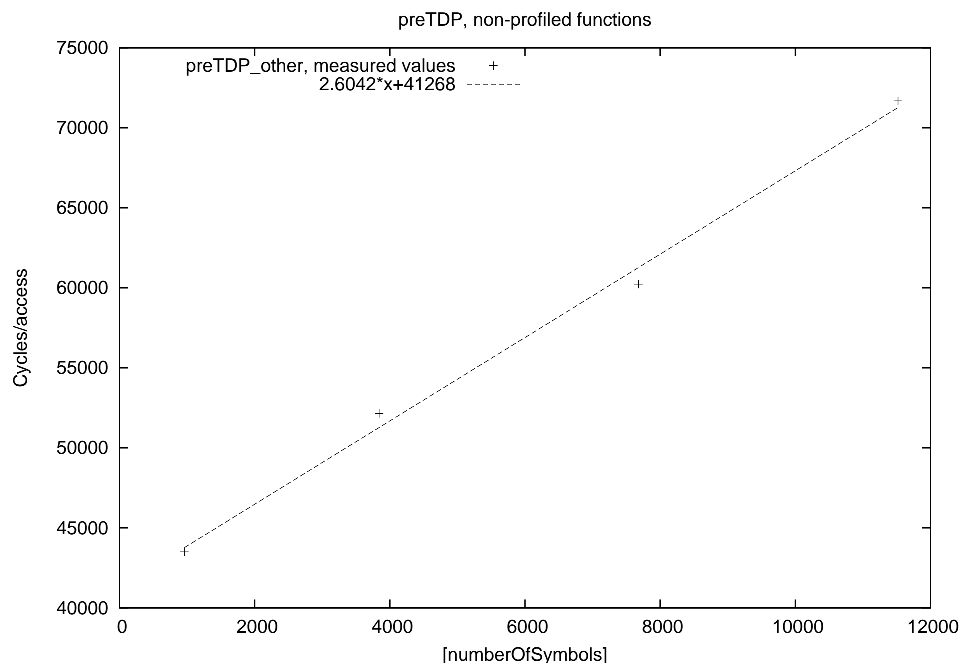

Figure 4.7: *A graph illustrating the correlation between* y = *the amount of clock cycles consumed by pre-TDP, non-profiled functions and* x = *the number of symbols being processed.*

Remaining costs

When collecting the empirical results, focus was directed to analyzing the costs which were of significant magnitude and varied depending on the type of input data being processed. The non-variable and relatively small costs can be combined into one large bulk of clock cycles, "preTDP_other" for our modeling purposes. Figure 4.7 contains a graph illustrating the clock cycles consumed by this block depending on the length of the data being processed. The conducted studies show that although the costs outside our six profiled functions were of the same magnitude, there was a small increase when trying more dense user data. This increase turned out to be linear, and it was therefore easy to derive a linear equation $y = kx + m$ (x = number of symbols being processed) to model the costs.

doPostTdpProc()

The function *doPostTdpProc()* depends on the following variables:

- Number of Symbols

The *doPostTdpProc()* function is responsible for the frame protocol building, i.e. packaging of user data before it leaves the system, and no signal processing is made here. Therefore, it was believed that the cycles consumed by this function would only depend on the amount of actual user data, i.e. the number of symbols. After plotting a graph by using cycles as y-values and symbols as x values and utilizing the least-squares method, a linear function was derived. The linear function of the form $y = kx + m$ gave very accurate results.

4.7.2 Turbo Decoder Peripheral

When calculating the processing delay of the TDP, the amount of cycles consumed between calling the function *startTDP()* and reaching an `if`-clause in the process *IO_DMA_intr* was measured. The results of these measurements are shown in figure 4.8. By utilizing a software queue, the TDP always works with one sub frame at the time. Each sub frame is divided into code blocks, and one code block at the time is processed. The code blocks are of a fixed size, and the size and the number of code blocks for a sub frame is decided by the ETFCI. The DMA transfers one code block at the time which the TDP processes. When a sub frame is processed, the DMA generates an interrupt that is caught by the DMA interrupt process. The processing time for the TDP was found to be linear to the code block size multiplied with the number of code blocks, and hence a linear equation of the form $y = kx + m$ could be derived.

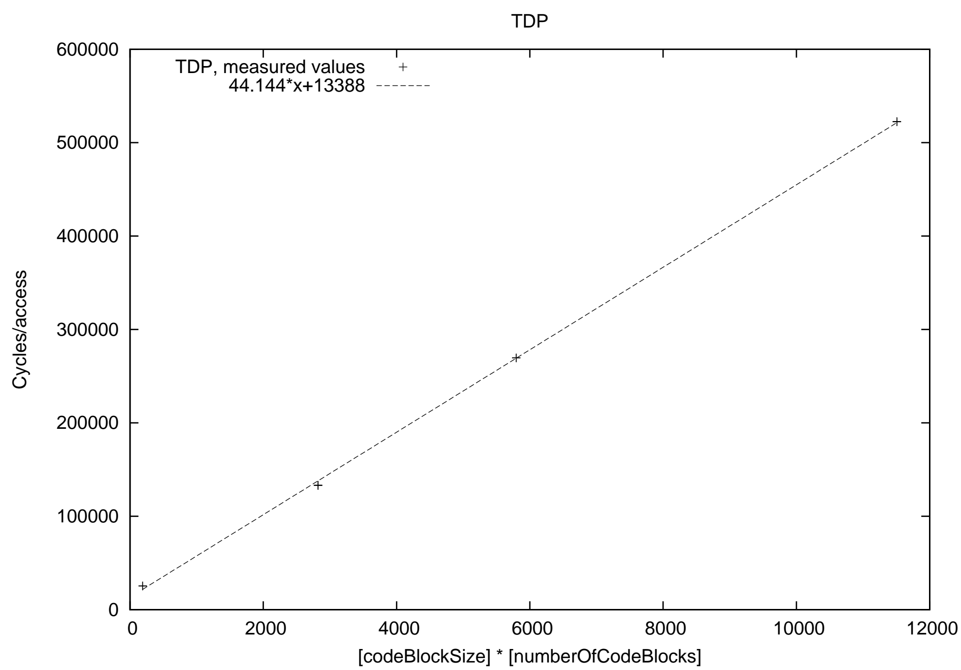

Figure 4.8: *A graph illustrating the connection between y = the delay (amount of clock cycles) from calling TDP to receiving completion signal and $x = [codeBlockSize] * [numberOfCodeBlocks]$.*

Function/Component	**Formula**	**Parameter(s)**
doScaleAndQuantization()	$SymbolType = 1:$ $z = 1.5948x + 4197.4y - 745$ $SymbolType = 2:$ $z = 2.8583x + 4197.4y - 745$	$x = [numberOfSymbols]$ $y = [numberOfChannels]$
doSecondDeInterleaving()	$y = 1.1176x + 189.64$	$x = [numberOfSymbols]$
doRateDeMatch()	$y = 6.1292x + 926.18$	$x = [numberOfCodeBlocks] * ([codeBlockSize] + 4)$
doLimitAndSegmentation()	$y = 4.798x + 20$	$x = [numberOfCodeBlocks] * ([codeBlockSize] + 4)$
calcTdpParams()	$z = 7535x + 12.054y + 3279$	$x = [numberOfCodeBlocks]$ $y = [codeBlockSize]$
RemainingCosts()	$y = 2.6042x + 41268$	$x = [numberOfSymbols]$
TDP	$y = 44.144x + 13388$	$x = [numberOfCodeBlocks] * [codeBlockSize]$

Table 4.2: *Formulas for calculating cycle consumption for different functions and components*

4.8 Chapter summary

In this chapter, the methodology for creating the model has been explained. The use of the ISS and verification against the target event logs as input source for the model has been described. The simulation library VirtualTime has also been presented, and an overview of the system and the different parts have been given. Also, the model's limitations and scope have been presented. For the functions identified to significantly influence the execution time and the TDP, formulas for calculating cycle consumption has been presented. These formulas are summarized in table 4.2.

Chapter 5

Verification

5.1 Introduction

In this chapter, the measurements and derived formulas from the ISS will be verified against the event log files, which were output from execution of the target system. The initial ISS measurements did not correspond to the event log files, and trying to find the cause of the inconsistencies between executing on target and in the ISS led to significant investigation work. Numerous factors affecting the measurements in the ISS were identified, these will be presented in this chapter. Also, output from execution of the model will be presented and compared with the corresponding flow in the target system.

5.2 Comparison between target and the instruction set simulator

Since the ISS is claimed to be an instruction accurate simulator it was believed that measurements performed in the ISS would give identical results compared to the target system. The ISS used is very flexible and a powerful profiling tool. It is possible to profile entire functions, loops, custom ranges etc. Because of this the original plan was to utilize

Function	**Cycles ISS**	**Cycles target**
doScaleAndQuantization()	335 129	10 272
doSecondDeInterleaving()	88 077	4 608
doRateDeMatch()	373 390	18 944
doLimitAndSegmentation()	117 388	6 016

Table 5.1: *Comparison between the instruction set simulator and target for test case 1, instruction set simulator code compiled with debug and no compile time optimizations*

Function	**Cycles ISS**	**Cycles target**
doScaleAndQuantization()	660 031	16 504
doSecondDeInterleaving()	173 667	9 008
doRateDeMatch()	771 432	30 088
doLimitAndSegmentation()	242 004	9 664

Table 5.2: *Comparison between the instruction set simulator and target for test case 2, instruction set simulator code compiled with debug and no compile time optimizations*

it as much as possible when performing the measurements, since it gives the possibility to measure on a very low level for formula derivation, and then verify the measurements against the target event logs. During the comparison it was discovered that many factors affected the results from the ISS. This led to many comparisons and changes before numbers from the ISS matched the numbers from the target system.

5.2.1 The instruction set simulator code compiled with debug and no compile time optimizations

Initially, of the four test cases used in the ISS, only two of them had been run on the target system utilizing event logs. Also, of the measurement points used in the ISS, only a few could be found at the same place in the target code. The consequences of this was that during this first comparison, it was only possible to compare the resource consumption for some of the functions, and only for two test cases.

Function	**Cycles ISS**	**Cycles target**
doScaleAndQuantization()	13 494	10 272
doSecondDeInterleaving()	4 486	4 608
doRateDeMatch()	18 037	18 944
doLimitAndSegmentation()	4 965	6 016

Table 5.3: *Comparison between the instruction set simulator and target for test case 1, instruction set simulator code compiled with debug and compile time optimizations (O=3)*

As can be seen in tables 5.1 and 5.2, the measurements are completely off, sometimes by a factor of $10 - 20$. Because the values from target are the actual values seen when running the system on real hardware, the values from the ISS were declared as being faulty. The values from the ISS were also unrealistic. When multiplying the total amount of cycles used to process one sub frame of user data with the total amount of sub frames the system should be able to handle per second, the resulting cycles were much larger than the CPU's frequency.

5.2.2 The instruction set simulator code compiled with debug and compile time optimizations

When checking the Makefile, it turned out that the binary, TB, used in the ISS was not compiled with any optimization flags. The target system and TB has different targets in the Makefile and the TB target lacked optimization flags. When these flags were added, completely different results were obtained. These results are presented in tables 5.3 and 5.4.

The results from the ISS were now of the same magnitude as the target results. It may seem a bit strange to the unversed that the optimization flags could do so much difference ($5 - 10\%$ of the initial cost), but this is to be expected and is well known. Using optimization significantly lessened the possibility for doing measurements. When stepping in the code during execution, the flow does not always correspond to the flow seen in the

Function	**Cycles ISS**	**Cycles target**
doScaleAndQuantization()	20 054	16 504
doSecondDeInterleaving()	8 972	9 008
doRateDeMatch()	37 175	30 088
doLimitAndSegmentation()	10 509	9 664

Table 5.4: *Comparison between the instruction set simulator and target for test case 2, instruction set simulator code compiled with debug and compile time optimizations (O=3)*

Function	**Cycles ISS**	**Cycles target**
doScaleAndQuantization()	13 781	10 272
doSecondDeInterleaving()	4 525	4 608
doRateDeMatch()	18 023	18 944
doLimitAndSegmentation()	4 930	6 016

Table 5.5: *Comparison between the instruction set simulator and target for test case 1, instruction set simulator code compiled without debug, with compile time optimizations (O=3)*

source code. The placement of breakpoints is also affected, and sometimes it is not even possible to profile smaller functions. Loops are often rolled out, and does not iterate as expected when looking at the source code. Despite this, the functions which were looked at when comparing results from the ISS and target were still possible to profile. However, the measurements were still inaccurate and the investigation continued.

5.2.3 The instruction set simulator code compiled without debug, with compile time optimizations

When running the TB in the ISS, compiling TB with debug flag gives the possibility to place breakpoints in the code at arbitrary places. This is important for measuring iteration costs for loops etc. The target system is however compiled without debug, and for making a fair comparison it was realized that the compiling options affecting performance must be equal. Also, it was suspected when using both optimization and debug, the two settings

Function	**Cycles ISS**	**Cycles target**
doScaleAndQuantization()	19 486	16 504
doSecondDeInterleaving()	8 798	9 008
doRateDeMatch()	37 189	30 088
doLimitAndSegmentation()	10 466	9 664

Table 5.6: *Comparison between the instruction set simulator and target for test case 2, instruction set simulator code compiled without debug, with compile time optimizations (O=3)*

tend to work against each other. TB was recompiled without debug which led to new results (tables 5.5 and 5.6). Using optimizations coupled with no debug gives very limited possibilities for measuring in the ISS. One is usually limited to profile larger functions only. The results were now of the same magnitude as the results from the actual system, but there was still a difference.

5.2.4 Discussion about the comparisons between target and the instruction set simulator

Even though the results from the ISS were now close to the target results, they still differed. Different theories why this occurred were discussed and investigated.

Different code base

Without going into too much detail, TB used a code base which is somewhat different to the code base used when running on target and outputting event logs. It was believed that some optimizations had been done to the data processing chain in the code base used for TB which could affect results slightly and resulting in a lower amount of clock cycles when measuring on TB in the ISS because of these optimizations. However, this could not be the complete truth as the results in target sometimes showed fewer cycles than the results from the ISS.

Differences in test cases

Even though the input data of the two pairs of compared test cases are of the same type, there is a difference that might influence the results. The test cases for TB are always single user while the test cases in target consisted of multiple users. At first, it was believed that this did not matter because the sub frames for each user is processed in series and not in parallel, one at a time in the application scheduler. When analyzing the log files, it was believed that as long as no context switches occurred during the processing of a sub frame, the other users should not affect the sub frame currently being processed. However, even if no context switch occurs during the processing of a sub frame, it was during discussion brought up that perhaps the access time to the external memory might be affected because of concurrent DMA jobs of other users. If the DMA currently is moving data belonging to another user, this should increase the latency when accessing the external memory during the data processing of the current sub frame. This theory was verified during profiling of function $calcTdpParams()$, see sections 4.7.1 and 6.4.4.

Settings in the instruction set simulator

When using the ISS, given an object to profile, i.e. a function, it is possible to measure the effective amount of cycles consumed by the CPU (*cycles CPU*) or the total amount of cycles (*cycles total*) consumed. For example, if a function takes 1000 cycles to execute, and only 500 of these cycles involves the CPU actually executing instructions to complete calculations. The remaining 500 cycles adhere from the latency caused from fetching data from external memory pipeline stalls. These two different ways of measuring cycles greatly influences the results. When looking at the target event logs, the time stamps corresponds to cycles total since the time elapsed between two events includes memory and other latencies. Thus, when doing comparisons between the ISS and target, it is important to look at cycles total in the ISS. Cycles CPU can however still give important information when identifying parameters that influence the actual computation time in a profiling

object.

Interference from the test bench code

Since the test bench code is compiled into the same binary as the regular system code, it runs on the same CPU and hence competes for resources with the regular system code. This should in theory affect the performance of the system because when the processes used by the test bench is swapped in, the regular system is halted. However, this was investigated and it can be said with a fair certainness that the test bench is swapped out after injecting user data to the system, and not swapped in before all the user data has been output. Hence, during the user data processing, the test bench is swapped out and does not interfere with the system. Since the measurements are performed during the user data processing, the test bench should have zero interference in our measurements.

5.2.5 Addressing the found issues

Getting the output from the ISS to match the target is of vital importance. Measuring in the ISS has as mentioned earlier many benefits. But in order to make a fair comparison, the possible causes for deviations must be eliminated and it was hence necessary to address them. By changing the code base and reducing user data for the test cases run in target to one single user, a more accurate comparison could be made. Changing the code base and test cases both involved a significant amount of work by the engineers working on the system. Because of the limited amount of time set aside for the project, verification was only made on two of the test cases. Additional log points were added to the code in order to enable profiling of the the TDP and all the functions that were profiled in the ISS. A debug flag was also added to the makefile since it is beneficial to have the debug flag when profiling in the ISS. Since the production target version runs without debug, profiling in reality must be done without debug in the ISS. However, at this stage only correspondence of target and the ISS is of interest.

Function	**Cycles ISS**	**Cycles target**	**Deviation**
doScaleAndQuantization()	13 494	9 816	37.5%
doSecondDeInterleaving()	4 486	4 656	-3.65%
doRateDeMatch()	18 037	19 056	-5.35%
doLimitAndSegmentation()	4 965	5 960	-16.7%
calcTdpParams()	44 447	46 080	-3.54%
doPostTdpProc()	33 059	31 168	6.07%
TDP processing	133 096	198 096	-32.8%

Table 5.7: *Final comparison between the instruction set simulator and after addressing the found issues for test case 1.*

Function	**Cycles ISS**	**Cycles target**	**Deviation**
doScaleAndQuantization()	20 054	15 872	26.3%
doSecondDeInterleaving()	8 764	9 088	-3.57%
doRateDeMatch()	37 175	38 296	-2.93%
doLimitAndSegmentation()	10 509	11 504	-8.65%
calcTdpParams()	51 983	55 464	-6.28%
doPostTdpProc()	44 639	43 736	2.06%
TDP processing	269 500	397 832	-32.3%

Table 5.8: *Final comparison between the instruction set simulator and target after addressing the found issues for test case 2.*

It was unfortunately not possible to make comparison of the remaining costs of the signaling processing done before the TDP (see section 4.7.1), because the main function for handling this processing is a bit different between TB and the target version. The signal processing is the same, but the method of preparing the user data for processing differs. A fair comparison was therefore not possible.

5.2.6 Final comparison

As seen in tables 5.7 and 5.8, when comparing ISS and target, the functions *doScaleAndQuantization()* and *LimitAndSegment()* differs significantly in terms of percentage while the other functions are closer. The formula used to calculate the deviations is

Function	Cycles model	Cycles target	Deviation
doScaleAndQuantization()	13 615	9 816	38.7%
doSecondDeInterleaving()	4 481	4 656	-3.76%
doRateDeMatch()	18 247	19 056	-4.25%
doLimitAndSegmentation()	5 101	5 960	-14.4%
calcTdpParams()	44 830	46 080	-2.71%
doPostTdpProc()	32 955	31 168	5.73%
TDP processing	137 962	198 096	-30.4%

Table 5.9: *Comparison between target and model for test case 1.*

$deviation = (cycles_{ISS} - cycles_{target})/cycles_{target}$ for tables 5.7 and 5.8. For tables 5.9 and 5.10 the formula used for calculating the deviations is $deviation = (cycles_{model} - cycles_{target})/cycles_{target}$.

For *doScaleAndQuantization()*, a theory explaining the deviation is that the signaling messages differs between the test cases for the ISS and for target. This causes a different execution flow to be taken in the function which has a different cost. Because of the limited time available, this theory was not investigated and is therefore unconfirmed.

The deviation sources for the function *doLimitAndSegmentation()* are unknown. The only suspicious finding is the use of a semaphore. The time for locking and releasing the semaphore might be different for currently unknown reasons. This theory needs further investigation.

The results of the TDP processing are significantly off, but with the same percentage for both test cases. This finding is discussed in section 5.3.1. The smaller deviations also needs further investigation. Theories will be presented in section 5.3.1.

5.3 Comparison between Model and target

As the model is based on the profiling results from the ISS, the comparison between the model and target shows similar results as the comparison between the ISS and target. As

Function	**Cycles model**	**Cycles target**	**Deviation**
doScaleAndQuantization()	19 581	15 872	23.4%
doSecondDeInterleaving()	8 772	9 088	-3.48%
doRateDeMatch()	36 500	38 296	-4.69%
doLimitAndSegmentation()	10 460	11 504	-9.08%
calcTdpParams()	53 281	55 464	-3.94%
doPostTdpProc()	44 683	43 736	2.17%
TDP processing	269 247	397 832	-32.3%

Table 5.10: *Final comparison between target and model for test case 2.*

seen in tables 5.9 and 5.10, the current model fails to fulfill the ±1% deviation requirement described in section 3.3.2. Reaching an accuracy of ±1% will be hard. Even with just four test cases, it is difficult to create a model (i.e. a formula) for calculating the costs that is accurate enough without being too complex. More test cases are needed in order to establish the fact whether it is possible to reach this accuracy level.

5.3.1 Further discussion about remaining deviations

Differences in test cases

Even though single user test cases for the target system were constructed, there could be other parameters of the user data besides the ETFCI that influences the signal processing and that are different from the test cases for the TB. Another explanation causing differences in the user data could be that in the TB, user data is injected directly into the decoder. In the target system, the user data is injected earlier in the telecommunication system and passes through other systems before it reaches the decoder. These system could perhaps change the attributes of the user data causing the decoder to behave differently than expected.

Hardware model in the ISS

The results of profiling the TDP were significantly inaccurate. Perhaps the simulation of the hardware is functionally correct, but not cycle accurate. If this is the case for the TDP, perhaps the DMA controller is also only functionally modeled.

Hardware frequencies

Some of the different hardware components in the system run on frequencies that are relative to the CPU frequency. In the ISS, it was only possible to individually select frequency of the CPU and the external memory bus. They were selected so that the ratio between the frequencies were identical to the target system. However, the individual frequencies were not identical to the target system. If the other hardware depends on these frequencies, they run at a different speed in the ISS compared to the target system.

Conclusion

The ISS is a practical tool for doing profiling, but it has still not been completely verified to be accurate. The deviations found during the final comparison in section 5.2.6 and the theories in section 5.3.1 need further investigations before the ISS can be claimed to be accurate enough for use as a profiling tool when creating input to the model. When making comparisons like done in this chapter, it is important to have control over the compilation flags. Different flags affects both the profiling results and the ability to perform accurate measurements when using an ISS.

Chapter 6

VirtualTime implementation

6.1 Introduction

In this chapter the implementation of the model in VirtualTime will be described. Each component will be examined and discussed. As the time for the thesis was limited, the model is not complete. The limitations of the model will be presented as well as future work.

6.2 Limitations

The model has a fair amount of limitations, both functional and accuracy related. The functional limitations have already been discussed in section 4.6. These are the functional limitations briefly summarized:

- Only EDCH traffic.
- Only 2ms TTI.
- Only user plane, control plane omitted (beneficial).
- Only the TDP and application scheduler have been studied.

- Retransmission of user data is not supported.

There are also limitations concerning the accuracy of the model:

- Only based on four test cases.
- Results from the ISS only verified against two test cases.
- Hardware model partly integrated in software model.
- The accuracy has not reached the ±1% deviation requirement.

The hardware model is very limited, and is partly integrated into the software model because of the use of *cycles total* when measuring. It is more proper to profile all the objects in the software with *cycles CPU* instead of *cycles total*, and then use a proper model for the DMA and external memory accesses in order to model the extra costs involved with external memory latency.

6.3 Overview

As VirtualTime is a C-library, the main program consists of a main function. In this function a CPU, log points and processes are created. Also, this is where settings like CPU-speed and simulation time are setup. The processes represents the different components and can be of different types depending on the nature of the component, e.g. a prioritized process, external hardware, etc. Each process is implemented as a regular C-function, where regular C-code can be used in conjunction with specific VirtualTime functions and types (prefixed with *vt_*) for using the virtual CPU, sending and receiving signals, etc.

6.4 Implementation of each component

In this section, the implementation of each component identified in section 2.4.1 is described. As mentioned in section 6.2, not all components are sufficiently modeled. All

```
void process(vt_process_t *me){
    vt_signal_t s;
    vt_sig_select_t sig_list[3] = {2, A_SIGNAL, ANOTHER_SIGNAL};
    TTI *tti_ptr;

    for ( ;; ){
        vt_sel_receive(&s, sig_list);

        switch(s.sig_no){
            case A_SIGNAL:
                handleASignal(); break;
            case ANOTHER_SIGNAL:
                handleAnotherSignal(); break;
        }
    }
}
```

Figure 6.1: *General component design in VirtualTime*

components consist of an infinitive loop which prevents the functions implementing the processes to end their execution. The general design of all implemented components is depicted in figure 6.1. For specific implementation details, please refer to the bundled source code.

6.4.1 User Data

The user data consists of a regular C-struct containing the different parameters found to influence the resource utilization in the system. A pointer to the struct is passed along with all the signals being sent between the different components, and the parameters of the struct are used to calculate costs in various parts of the system. This abstraction of user data is very powerful as it is easy to generate new user data, and to add new parameters found to influence the system when the model is refined. Future work of the user data probably involves the addition of new parameters. Information about timing could perhaps be derived from the user data.

6.4.2 Frame buffer

The important characteristics of the FB are that it is an external hardware and generates interrupts to signal that new user data is available. Since it is an external hardware, it is modeled as an external process which does not consume CPU cycles. Also, it is in the FB that the used data is created. The timing of the interrupts can easily be changed by using the call *vt_delay_until_time()* in conjunction with loops. The interrupt is modeled with a signal which is sent to the FB ISR. The signal contains a pointer to the C-struct representing the user data. Improvements and future work of the FB component involves reading the user data from a file, it is now created in code. A standardized timing control should be decided, in conjunction with the user data as mentioned in section 6.4.1.

6.4.3 Frame buffer interrupt service routine

An interrupt service routine differs from a prioritized process in the fact that it is not necessary to do a full context switch when doing a swap. This distinction is important because the cost for swapping in a ISR is less then swapping in a regular process. Therefore, the FB ISR is modeled as an interrupt process in VirtualTime. The functionality consists of starting a DMA job to transfer the user data from the FB to the internal memory. At the top of the infinitive loop which all VirtualTime processes have, there is a call to *vt_receive()* which makes the process wait for a signal. As soon as the FB sends a signal, the process is swapped in. The signal contains a pointer to the user data, and new signal is then sent to the DMA, also containing this pointer.

The FB ISR has not been profiled, so the amount of clock cycles consumed is unknown. But as no data processing is made and the copying of user data to internal memory is performed by the DMA, the amount of cycles are believed to be fairly constant. Deviations may occur when starting the DMA job because of other ongoing DMA jobs, but that must be modeled in the DMA. Future work hence involves profiling to find this, probably mostly

static, cycle cost.

6.4.4 DMA

The DMA controller is believed to be the most advanced component in the whole system. The implementation of this component is limited, and is only made to the point where the flow and interaction of the different components of the system is correct. It is modeled as an external process in VirtualTime, as it is an external hardware that does not use the CPU. It is modeled with the obligatory infinitive loop, which waits for the arrival of a signal. The signal describes the job to be performed by the DMA, but the amount of time used to complete these different jobs has not been profiled. When a job is finished, the DMA generates an interrupt which is caught by the DMA interrupt process. This is modeled as sending a signal to the VirtualTime process implementing that process.

The future work in modeling the DMA involves identification of the different types of jobs, and profiling these job types to derive how the parameters, e.g. size of the job, influences the time for completion. Also, the queue management of the jobs must be identified, i.e. how jobs are split into smaller pieces and which jobs have priority over other jobs.

It is believed that the modeling of the DMA must be accurate, because it introduces significant delay when the CPU stalls while trying to access the external memory during an ongoing DMA job. Also, all parts in the code where external memory access takes place must be identified and placed in the model since these are places in the code where the CPU may stall for different amount of time depending on the current DMA load. The memory/cache hierarchy is depicted in figure 6.2.

6.4.5 DMA interrupt process

In VirtualTime, a prioritized process is used for modeling this process. It runs with a higher priority than the application scheduler, so in case an interrupt occurs during work being

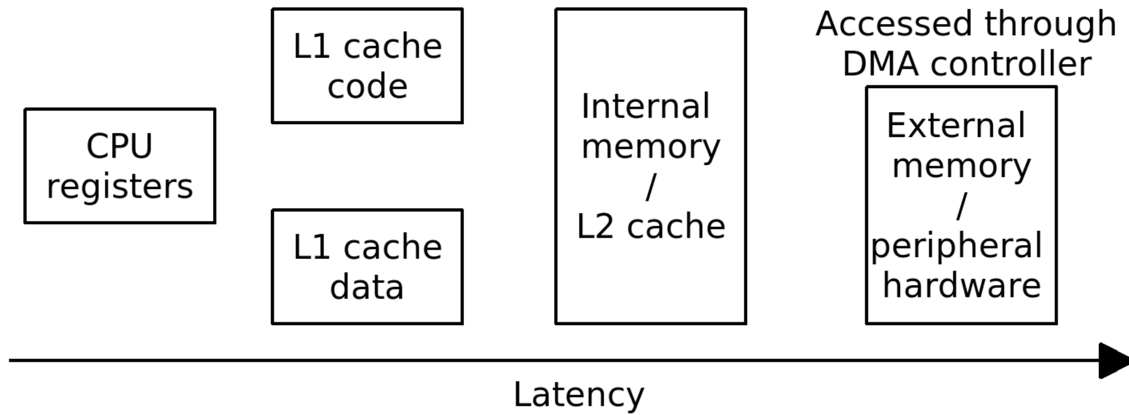

Figure 6.2: *A figure illustrating the memory/cache hierarchy of the studied system.*

performed by the application scheduler, the DMA interrupt process gets swapped in. It carries the standard design of an infinitive *for* loop, waiting for a signal. It currently receives two signals *FRAME_BUFFER_JOB_COMPLETED* and *TDP_JOB_COMPLETED*. After the receiving of these signals, a signal is sent to the application scheduler.

This process is only implemented to get the correct application flow. It has not been profiled, so future work involves populating the implementing function with correct cycle usage. The number of signals/interrupts received is also limited and additions are needed.

6.4.6 TDP

The TDP is an external hardware which is implemented in VirtualTime as an external process not utilizing the CPU, it only consumes time. Since the queue management for the TDP is placed in the application scheduler, the TDP itself does not manage any queue and only does one job at a time. The delay of the processing is decided by the amount of code blocks and the code block size of the user data. After completion, a signal is sent to the DMA controller indicating that the job is completed.

When the TDP was profiled, the delay between the function *startTdp()* and the receiving of an interrupt in the DMA interrupt process was measured. This delay does not solely

depend on the TDP, but also the DMA controller which sets up the TDP by chaining some DMA jobs. The delay needs to be split up between the TDP and DMA, with correct portions being placed on respective component.

6.4.7 Application Scheduler

The application scheduler process is where most of the analyzing work was done, since it does the majority of the user data processing. It is implemented as a regular prioritized process in VirtualTime. It listens for signals and launches a sub function depending on the type of signal received. Since the model is limited to 2 ms TTI EDCH traffic and the control plane is omitted, it only handles two signals; *DATA_FB_READY* and *DATA_TDP_READY* which takes care of the user data processing before the TDP and the packaging of user data after the TDP job has completed, respectively. The function for handling the *DATA_FB_READY* signal calculates the amount of cycles consumed in six different portions, the five functions and remaining costs identified in section 4.7.1. The user data struct is used to derive the parameters necessary to calculate the cost for each of the six steps.

The future work of modeling the application scheduler involves verification and refinement of the formulas used to calculate the costs by obtaining measurement results from more test cases. Correct queue handling is also necessary in order to get accurate results. The queue handling in the model is simplified into using a simple OSEck signal queue. In reality, the OSEck signal queue is constantly drained of signals which are placed into a different custom queue implementation depending on the type of signal. This method is used to shorten the time it takes to find a signal of a certain type. Finally, and perhaps most importantly, all DMA jobs and external memory accesses need to be placed at correct places in the functions. This is predicted to be a both challenging and critical task.

Chapter 7

Discussion

7.1 Introduction

A discussion about the thesis work will be presented in this chapter. The model choice will be evaluated together with the simulation library VirtualTime. Two aspects found to be of significant importance when creating a model will be presented as well as a summary of future work on the modelling.

7.2 The model choice

During the feasibility study in chapter 3, the abstract model was chosen although several risks had been identified. The risks had been identified as failing to reach sufficient accuracy and also that the amount of time might be insufficient considering the amount analyzing work needed. The choice was made because an accurate model of this type would be more valuable as compared to the target code model, and also because the modeling process itself would provide valuable information even if a final model could not be created within the limited amount of time available for the thesis. These assumptions have been proven to be correct. There are indeed several problems when creating a complete abstraction of

a system, and there was and is a lot to be learned from this kind of process. The identified important aspects when modeling are found in section 7.4.

7.3 Simulation library

Not much time was spent on the actual modeling in VirtualTime, as compared to analyzing the system and verifying measurements from the ISS. It is therefore hard to draw any conclusions about how VirtualTime performs when the model complexity increases. But for the level of complexity reached in the model of this thesis, VirtualTime has performed well and did not give any problems of significance. If VirtualTime works as the specifications say it should, it is a very powerful and suitable tool for creating abstract models of complex real-time systems. The types and functions defined by VirtualTime are few and straight forward, but the combination with regular C-code makes it an easy and at the same time powerful tool when creating complex models.

7.4 Important aspects when modeling

When creating an abstract model, there are mainly two things to consider: System knowledge and efficient profiling. These two corner stones of the modeling process will now be summarized.

7.4.1 System knowledge

In order to achieve efficient modeling, great knowledge about the system must be gained. This knowledge can help when deciding the abstraction level of the different components, and also when identifying the different components. Some components, perhaps with a fixed execution time, can be very easy to model. Other parts, like the processing steps in the studied system, requires knowledge to help estimate what parameters influences

the execution time. During this thesis, a significant amount of time was spent just to understand the system.

7.4.2 Profiling

An abstract model is always based on having measurements serve as references when populating the model with delays and resource utilizations. It is important to have efficient profiling in terms usability, amount of test cases, accuracy and performance. Since obtaining sufficient knowledge about the system is vital for an accurate model, many test cases must be run many times, and the results of these must be examined. If the profiling process is tedious and takes time, the modeling process suffers. If the model is only derived from a couple of test cases, its robustness and accuracy will suffer. Also, the results obtained from the profiling must be trustworthy. It is of significant importance to know what is measured and if other hidden factors can influence the results.

Using a simulator can be preferable since being able to start, stop and debug during execution is often beneficiary. If using a simulator, it is important to verify that measurements performed in the simulator corresponds to the same measurements performed in the target system. The verification of the simulator results was a significant part of the practical work of this thesis.

7.5 Model future work

Details about the limitations and future work of the modeling can be found in chapter 6, this section serves as a summary.

The user data processing chain has been closely examined and profiled. However, the formulas derived which describes the cycle consumption need further verification using additional test cases. When using few test cases, there is a great risk that incorrect attributes of the user data are chosen as parameters for the functions.

The model also lacks a proper hardware model. Since the formulas are based on a measuring cycles total (i.e. including external memory latency, see section 5.2.4), the hardware's contribution to the latency is partly integrated into the software model. The DMA controller needs to be accurately modeled, and all external memory accesses must be put into the software model. The software model also needs more work in some smaller components, like the FB ISR and the DMA interrupt process.

The work of identifying all DMA jobs and external memory accesses is a very important task for reaching sufficient accuracy. This will probably be a very tedious task, even for engineers having good knowledge of the system. This finding also spawned the theory that the actual system might need a re-design of the DMA and external memory usage. If something is too hard to model, perhaps the design is too complex and needs reworking.

Since the DMA and external memory usage have a large impact on the latency coming from CPU stalls, which is hard to predict, a redesign of how the DMA and the external memory are used could lessen this almost unpredictable behavior and also make the modeling process easier. Since the CPU stalls have shown to be a large problem in the system, this is probably a good path to choose since it benefits both the model creating process and the overall performance of the system. With a good hardware model, it should also be possible to see how concurrent users affect each other.

Chapter 8

Conclusion

In order to provide benefits over just measuring on a real-time system some parts of the system need to be abstracted, i.e. a model must be created. The creation of such a model involves a lot of work and places high requirements on system knowledge among the people involved in the model creation process. There is also a need for efficient profiling of the system in regards to usability, amount of available test cases, accuracy and performance. Without proper knowledge it is hard to decide a proper abstraction level for different components, and also hard to find the attributes that influences the execution time and latency of the components. Efficient profiling is needed for generating input to the modeling process, but also for verification which is of major importance if the model is to be trusted.

During the work of this thesis, problems were encountered regarding both system knowledge and profiling. The limited knowledge and the profiling problems significantly slowed down the actual model implementation, but led to interesting findings about both the system and the modeling process itself. If something is too hard to model, perhaps the design itself needs to be reworked. The modeling process alone can therefore lead to important insights about the studied system, even before the creation of a complete model has been finished. For the actual system studied in this thesis, it was learned that the usage pattern of the DMA controller and external memory were complex and might need redesigning.

References

[1] 3GPP TS 25.201 version 6.0.0 Release 6; Physical layer - General description. 3GPPs website, December 2003. http://www.3gpp.org.

[2] 3GPP TS 25.212 version 6.2.0 Release 6; Multiplexing and channel coding (FDD). 3GPPs website, June 2004. http://www.3gpp.org.

[3] 3GPP TS 25.321 version 6.2.0 Release 6; Medium Access Control (MAC) protocol specification - annex A (normative). 3GPPs website, June 2006. http://www.3gpp.org.

[4] Johan Andersson, Anders Wall, and Christer Norström. Validating temporal behavior models of complex real-time systems. In Proceedings of the Fourth Conference on Software Engineering Research and Practice in Sweden (SERPS'04), September 2004.

[5] Johan Andersson, Anders Wall, and Christer Norström. Validating timing models of industrial real-time systems. Technical Report, Mälardalen University, June 2004.

[6] Erik Axling. Automatic generation of simulation models from designs. Master's thesis, Linköping University, Department of Computer and Information Science, 2007.

[7] ENEA. ENEAs website, December 2008. http://www.enea.com/.

[8] Andreas Ermedahl. A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, 2003.

[9] Hermann Kopetz. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 2002.

[10] Johan Kraft, Joel Huselius, Anders Wall, and Christer Norström. Extracting simulation models from complex embedded real-time systems. In Real-Time in Sweden 2007, August 2007.

[11] OSEck. ENEAs website, December 2008. http://www.enea.com/templates/Extension____12766.aspx.

[12] Rapita Systems. Rapita Systems website, December 2008. http://www.rapitasystems.com/.

[13] Mohammed El Shobaki. On-chip monitoring for non-intrusive hardware/software observability. Licentiate thesis, September 2004.

[14] Abraham Silberchatz, Greg Gagne, and Peter Baer Galvin. Operating System Concepts. Wiley, 2005.

[15] Universal Mobile Telecommunications System. 3GPPs website - UMTS, December 2008. http://www.3gpp.org/article/umts.

[16] VirtualTime. RapidaSystems website, December 2008. http://www.rapitasystems.com/virtualtime.

[17] Anders Wall, Johan Andersson, Jonas Neander, Christer Norström, and Martin Lembke. Introducing temporal analyzability late in the lifecycle of complex real-time systems. In In proceedings of RTCSA 03, February 2003.

[18] Logic analyzer. Wikipedia, December 2008. http://en.wikipedia.org/wiki/Logic_analyzer.

[19] Probe effect. Wikipedia, December 2008. http://en.wikipedia.org/wiki/Probe_effect.

Glossary

3rd Generation Partnership Project A project uniting telecommunication standards bodies.

Application Programming Interface For the purposes of this thesis, a set of functions, procedures, methods and types provided by a library.

Central Processing Unit A machine that can execute computer programs.

Dedicated Channel User data type which is mainly used for encoding speech and lower data rates.

Direct Memory Access A feature that allows certain hardware subsystems to access system memory independently of the central processing unit.

Enhanced Dedicated Channel User data type which is mainly used for encoding high data rates.

Enhanced Transport Format Control Indicator Describes the user data properties, one of these is the user data length.

Frame Buffer A hardware component used to buffer data before it goes into the decoder.

Frame Protocol The protocol used for sending data from the decoder to the RNC.

Instruction Set Simulator A simulation model which mimics the behavior of the studied system.

Interupt Service Routine A procedure which is run when receiving an interrupt.

Millions of Cycles Per Second A measure for processing power in computers.

Node B A base station in an UMTS radio access network.

Operating System Embedded compact kernel A real-time operating system designed to be run on embedded systems.

Radio Network Controller The governing element in the UMTS radio access network responsible for control of the Node Bs which are connected to the controller.

Test Bench For the purposes of this thesis, a limited environment for running profiling test cases in an instruction set simulator.

Transmission Time Interval A parameter in UMTS, refers to the length of an independently encodable transmission on the radio link.

Turbo Decoder Peripheral A co-processor used by the decoder to decode turbo encoded user data.

Universal Mobile Telecommunications System One of the third-generation cell phone technologies.

User Equipment A cell phone or other type of client.

Wideband Code Division Multiple Access A technology used to implement and realize UMTS.

Worst Case Execution Time The maximum length of time a task could take to execute on a specific system.

Acronyms

3GPP 3rd Generation Partnership Project. 9, 11

API Application Programming Interface. 43

CPU Central Processing Unit. 8–10, 20, 22–24, 27–31, 33, 39–43, 45, 55, 63, 66, 67, 71, 77, 78, 84

DCH Dedicated Channel. 46

DMA Direct Memory Access. 11, 13, 14, 29, 30, 39, 49, 55, 58, 66, 71, 74, 76–79, 84, 85

EDCH Enhanced Dedicated Channel. 11, 28, 46–49, 73, 79

ETFCI Enhanced Transport Format Control Indicator. 40, 58, 70

FB Frame Buffer. 11, 14, 49, 76, 79, 84

FP Frame Protocol. 14

ISR Interupt Service Routine. 11, 14, 76, 84

ISS Instruction Set Simulator. 32, 34, 38–40, 50, 55, 60–69, 71, 74, 82

MCPS Millions of Cycles Per Second. 24

OSEck Operating System Embedded compact kernel. v, 10, 41, 79

RNC Radio Network Controller. 10

TB Test Bench. 39, 40, 63–66, 68, 70

TDP Turbo Decoder Peripheral. 13, 14, 28, 30, 39, 47, 49, 57, 58, 60, 67–71, 73, 78, 79

TTI Transmission Time Interval. 11, 47–49, 73, 79

UE User Equipment. 10, 23

UMTS Universal Mobile Telecommunications System. 9

WCDMA Wideband Code Division Multiple Access. v, 9

WCET Worst Case Execution Time. 2