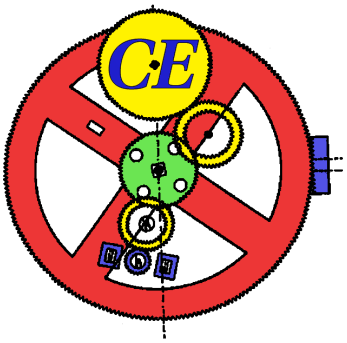


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

Execution time analysis of audio algorithms

Namitha Gopalakrishna

CE-MS-2014-11

Abstract

Execution time analysis forms an important part of design space and hardware architectural exploration for hard real-time systems. Some approaches for execution time analysis require prerequisite knowledge of synchronous data flows or timed automata employed in model checkers. This is unsuitable for industry-level use as easy approaches without any prerequisite knowledge requirements are preferred by them. Most of the existing proposals addressing execution time analysis target 'fast' approaches rather than timing accuracy. Also, these existing proposals are validated with simple processors as majority of processors used for hard real-time embedded software are not so complex. However, for audio applications employing audio algorithms, modern processors with high performance are required, to be scalable with increasing audio algorithm complexity. Execution time analysis of audio algorithms on these modern processors is gaining importance as sampling frequencies are getting higher and deadlines getting shorter. In this thesis, we propose an industry acceptable model/simulation framework to perform execution time analysis of audio algorithms on modern mono core and multi-core processors (operating in asymmetric multiprocessing mode). Our framework combines open-source tools such as Gcov, POOSL modelling language (Parallel object-oriented specification language) and Gem5 which is a computer architecture simulator to determine WCET of an audio algorithm using dynamic means as static WCET techniques lead to huge overestimations. This solution framework was proposed after conducting experiments targeting execution time analysis at different abstraction levels and evaluating results based on accuracy, flexibility, hardware compatibility and scalability. Our proposed technique is flexible as Gem5 models several processor architectures and expressing audio algorithms at the basic block abstraction level allows parameters such as loop bounds to be changed easily. It can be easily extended to more cores using the expressive POOSL modelling language. Our technique requires no prerequisite knowledge of any concepts and also solves issues such as influence of execution contexts of basic blocks on execution time in an implicit way without requiring additional analysis. We were able to achieve almost 99% timing accuracy for integer and floating point programs without long latency instructions using Gem5. For programs with long latency instructions, minor manipulations are presented which yielded almost 90% accuracy using Gem5.

TU Delft

Faculty of EEMCS

Execution time analysis of audio algorithms

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Namitha Gopalakrishna
born in Bangalore, India

Computer Engineering
Department of Electrical Engineering
Faculty of EEMCS
Delft University of Technology

Execution time analysis of audio algorithms

by Namitha Gopalakrishna

Abstract

Execution time analysis forms an important part of design space and hardware architectural exploration for hard real-time systems. Some approaches for execution time analysis require prerequisite knowledge of synchronous data flows or timed automata employed in model checkers. This is unsuitable for industry-level use as easy approaches without any prerequisite knowledge requirements are preferred by them. Most of the existing proposals addressing execution time analysis target 'fast' approaches rather than timing accuracy. Also, these existing proposals are validated with simple processors as majority of processors used for hard real-time embedded software are not so complex. However, for audio applications employing audio algorithms, modern processors with high performance are required, to be scalable with increasing audio algorithm complexity. Execution time analysis of audio algorithms on these modern processors is gaining importance as sampling frequencies are getting higher and deadlines getting shorter. In this thesis, we propose an industry acceptable model/simulation framework to perform execution time analysis of audio algorithms on modern mono core and multi-core processors (operating in asymmetric multiprocessing mode). Our framework combines open-source tools such as Gcov, POOSL modelling language (Parallel object-oriented specification language) and Gem5 which is a computer architecture simulator to determine WCET of an audio algorithm using dynamic means as static WCET techniques lead to huge overestimations. This solution framework was proposed after conducting experiments targeting execution time analysis at different abstraction levels and evaluating results based on accuracy, flexibility, hardware compatibility and scalability. Our proposed technique is flexible as Gem5 models several processor architectures and expressing audio algorithms at the basic block abstraction level allows parameters such as loop bounds to be changed easily. It can be easily extended to more cores using the expressive POOSL modelling language. Our technique requires no prerequisite knowledge of any concepts and also solves issues such as influence of execution contexts of basic blocks on execution time in an implicit way without requiring additional analysis. We were able to achieve almost 99% timing accuracy for integer and floating point programs without long latency instructions using Gem5. For programs with long latency instructions, minor manipulations are presented which yielded almost 90% accuracy using Gem5.

Laboratory : Computer Engineering
Codenumber : CE-MS-2014-11

Committee Members :

Advisor: Dr.Zaid Al-Ars, CE, TU Delft

Chairperson: Prof.dr.ir.Koen Bertels, CE, TU Delft

Member: Nick Hoogland, Software Architect, Bosch Security Systems

Member: Dr.Mathijs de Weerdt, Assoc. Professor, Algorithmics, TU Delft

Contents

List of Figures **vii**

List of Tables **ix**

Acknowledgements **xi**

1 Introduction **1**
1.1 Problem statement . . . 2
1.2 Motivation . . . 2
1.3 Approach and thesis contributions . . . 3
1.4 Thesis outline . . . 4

2 Background **5**
2.1 Audio signal processing . . . 5
2.1.1 Signal flow . . . 5
2.1.2 Requirements of real-time audio applications . . . 6
2.2 Modern Processors . . . 7
2.3 Execution time analysis . . . 8
2.3.1 Need for timing analysis . . . 9
2.3.2 Traditional methods of execution time estimation . . . 9
2.3.3 Timing Abnormalities . . . 10
2.3.4 Summary . . . 12

3 Execution time estimation at different abstraction levels **13**
3.1 Introduction . . . 13
3.1.1 What is a model? . . . 13
3.1.2 What are the advantages of modelling? . . . 13
3.1.3 Challenge of modeling . . . 14
3.2 High-level modelling techniques . . . 14
3.2.1 SDF (Synchronous Data Flow) . . . 14
3.2.2 POOSL modelling language . . . 16
3.3 Basic block methodology . . . 18
3.3.1 Static methods . . . 18
3.3.2 Hybrid methods . . . 19
3.4 Instruction level functional simulation (Loosely timed models) . . . 22
3.4.1 Dynamic binary translation (DBT) - QEMU . . . 22
3.4.2 Dynamic binary instrumentation (DBI) - Valgrind . . . 22
3.4.3 Fast Processor models - OVP . . . 23
3.5 Cycle accurate methods . . . 23
3.5.1 Hardware measurements . . . 23

3.5.2 Commercial solutions . . . 23
3.6 Summary . . . 24

4 Algorithmic abstraction level using POOSL **27**
4.1 POOSL vs SDF . . . 27
4.2 Hybrid approach using hardware and POOSL - algorithmic abstraction level . . . 28
4.2.1 Hardware/SDK setup . . . 28
4.2.2 POOSL setup . . . 29
4.2.3 The Approach . . . 31
4.3 Results . . . 32
4.4 Evaluation . . . 33

5 Basic block abstraction level **35**
5.1 Introduction . . . 35
5.2 Cycle accurate simulator - Keil . . . 37
5.3 Algorithm modelling at basic block level in POOSL . . . 37
5.3.1 Approach . . . 37
5.3.2 Advantages . . . 38
5.4 WCET analysis using aiT . . . 39
5.4.1 Static Analysis . . . 39
5.4.2 aiT working phases . . . 39
5.4.3 Manual annotations . . . 40
5.5 Results and Observations . . . 42

6 Cycle-accurate abstraction level **43**
6.1 Introduction . . . 43
6.2 Background and working . . . 44
6.2.1 Gem5 options . . . 45
6.2.2 Important features . . . 46
6.3 Configuration . . . 46
6.3.1 pipeline, cache and TLB configuration . . . 47
6.3.2 Functional units, operation latencies and branch predictor . . . 48
6.4 Fine tuning . . . 48
6.5 Experimental observations . . . 48
6.6 Results . . . 49
6.7 Floating Point errors . . . 51
6.8 Major differences between Gem5 and cortex-A9 . . . 51
6.8.1 Memory system . . . 52
6.8.2 Branch prediction and Replacement policies . . . 52
6.9 Summary . . . 53

7 Conclusions **55**
7.1 Main thesis contribution . . . 55
7.2 Comparison with related work . . . 57
7.3 Critical analysis . . . 58

7.4 Future work 59
7.5 Overall Conclusions 60

Bibliography **62**

8 Appendix **63**
8.1 Appendix A 63
8.2 Appendix B 69

List of Figures

1.1 Public addressing system. Source: Bosch Security Systems 2
1.2 Conference systems 2

2.1 Signal flow in audio applications 5
2.2 Ping-pong buffer scheme 6
2.3 Example of long timing effects. Source: [1] 11
2.4 Example of Cache timing anomaly. Source: [2] 11

3.1 Data Flow Graph. Source: [3] 14
3.2 Actor firing. Source: [3] 14
3.3 A snapshot of GUI showing different POOSL classes. Source: [4] 17
3.4 Comparison of different abstract levels for model of computation. Source: [5] 21

4.1 Algorithm block diagram 28
4.2 Zedboard design flow 29
4.3 Top-level system design 30
4.4 Application cluster 30
4.5 Processor cluster 30
4.6 Process class of Processor 31
4.7 Overview of the communication in POOSL 33

5.1 Cortex-M3 pipeline. Source: [6] 37
5.2 A part of the fft function divided into basic blocks 38
5.3 Control flow graph of basic blocks of the part in 5.2 38
5.4 aiT workflow. Source: AbsInt Angewandte Informatik GmbH, Safety Manual for aiT, Revision 217166 of April 2, 2014, used with permission 39
5.5 WCET using aiT for fft size 32 40

6.1 Workflow of gem5. Source: [7] 44
6.2 Cortex-A9 pipeline. Source: [8] 47
6.3 Execution Cycle count with error percentages on top of the graph 49
6.4 Branch misses in HW and Gem5 50
6.5 Execution cycle count on HW and Gem5 with min and max error percentages corresponding to manipulated and correct load store queue entries respectively mentioned on top of the graph 52

7.1 Model/simulation framework for dynamic WCET 56

List of Tables

4.1 Results of Algorithmic abstraction level 33

5.1 Results of Basic Block 42

6.1 Configuration changes for pipeline, cache and TLB 47
6.2 DRAM parameter changes 48
6.3 Execution cycle count for fixed point fft function 50
6.4 Speculative load stores and branch misses for fixed point fft 50
6.5 Execution cycle count for IIR-Gain-Limiter and simple float division program 50
6.6 Execution cycle count for IIR-Gain-Limiter and simple float division program with manipulated load store queue entries 50

7.1 Comparison of thesis with related work 58

Acknowledgements

I am very grateful to Hans van der Schaar at Bosch Security Systems for giving me this wonderful thesis opportunity to learn and contribute. This thesis would have been impossible without the cheerful guidance from my supervisor, Nick Hoogland, who was always there to motivate me when the going got tough and lend a patient ear at all our meetings. I would also like to thank Micheal and Patrick for sharing their valuable time and knowledge.

I would like to express my deepest and sincere gratitude to my professor and thesis guide, Dr. Zaid Al-Ars for being extremely motivational and inspiring even though we were in different cities. Thank you Dr. Zaid for being extremely patient with all my queries and answering my calls even if it was a Friday night! Special thanks to Dr. Mathijs de Weerdt for accepting the invite to be on the assessment committee.

Thank you Alba Magallon Baro, for being my idol and accompanying me on all our 'intense' study hours at the library. Thank you Miguel, Dario, Clara, Constantine and Max for being the best housemates ever. Thank you Ajay for always being there and making up for the absence of our beloved friends. Thank you Saurav for bearing with me through all the stressful times. Special thanks to my senior and well-wisher, Preethi Ramamurthy, for introducing me to Hans van der Schaar.

Last but never the least, thanks to my wonderful family for supporting my decision to come abroad for studies and providing the much needed emotional support. Thank you Mom, for being my pillar of strength. Hope reading this thesis brings you enough joy to make up for the sorrow you felt when I left your nest.

Namitha Gopalakrishna
Eindhoven,Netherlands
October 5, 2014

1 Introduction

Execution time analysis of modern real-time embedded systems for assuring timing guarantees is becoming more complex with each passing day. This complexity can be directly attributed to modern processors. Conventional methods to conduct performance analysis is no longer valid for modern processors as they possess a large number of dynamic features which leads to non-determinism and inaccuracy. Therefore, the industry seeks accurate yet easy-to-use methods for execution time analysis especially in the architectural exploration stage. The modelling and simulation tools industry is trying to address this problem and is developing methods to aid hardware/software co-design and analysis. Until now, most of these methods are commercial, therefore an active research community exists which is trying to solve the problem of execution time analysis with open-source tools.

Our team at Bosch Security Systems, which develops audio and conference systems involving transmission of voice, sound or music, also seeks answers to questions related to execution time analysis because they develop applications with real-time response requirements. Figure 1.1 is a block diagram of the Public Addressing system usually employed in buildings, airports, supermarkets etc. It consists of three components: call station, network controller and power amplifier. The target area where the announcements or messages need to be heard are divided into zones. The call station is used by the speaker to make announcements and also store pre-recorded messages for different zones. Based on the zone selection, the network controller routes either speech, music or sounds to the appropriate power amplifier. The power amplifier amplifies the output before sending it out through speakers in the respective zones. The components which perform audio signal processing are the call stations and power amplifiers. The network controllers are responsible for control and routing logic. The public addressing systems can also be used for evacuation purposes in buildings. Hence, real-time response of the systems becomes paramount in systems such as the voice evacuation systems.

Figure 1.2 shows the block diagram of a conference system developed at Bosch Security Systems. The wireless access point is the heart of the system where majority of the audio processing takes place. It also performs routing and control. The wireless devices are provided to the end users of the conference systems. A minor portion of audio signal processing takes place in the wireless devices as well. The conference systems need to adhere to certain quality of service requirements hence these systems have real-time constraints.

Thus applications at Bosch Security Systems consist mainly of control functionality and audio/video signal processing. With this knowledge of Bosch Security Systems products with real-time response requirements, we can now describe our problem statement

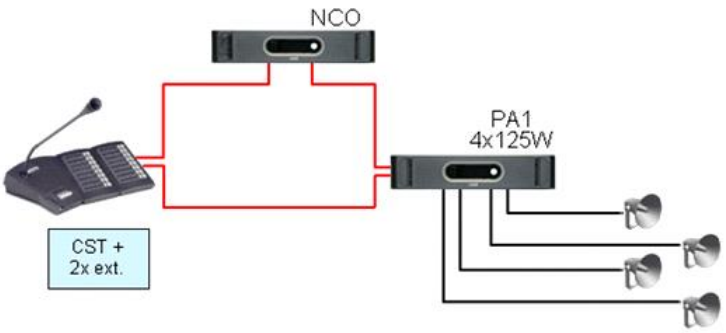


Figure 1.1: Public addressing system. Source: Bosch Security Systems

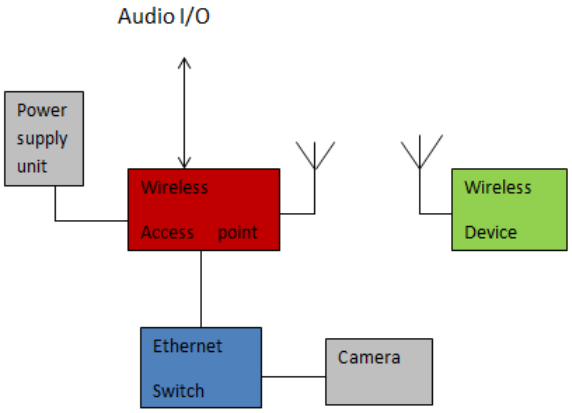


Figure 1.2: Conference systems

in the next section.

1.1 Problem statement

Our team at Bosch Security Systems has traditionally used digital signal processors for audio signal processing and general purpose processors for control related functionality. However, they would like to evaluate the feasibility of using modern general purpose processors for audio signal processing. Audio algorithms typically have large number of floating point operations and hard real-time deadlines. It is crucial to choose suitable hardware such that the real-time constraints of the application are met. Thus, this calls for an accurate method to determine program execution time of audio algorithms on modern general purpose processors. Another requirement is that the method should be generic and easy-to-use. The method should work for single core and multi-core processors when working in AMP (Asymmetric multiprocessing) mode such that one core is running Linux and executing a control application and other cores are executing au-

dio algorithms either bare-metal or with a RTOS (Real Time Operating System) if the algorithm is multithreaded.

Hence our problem statement can be formulated as 'How can we determine program execution time of audio algorithms of varying complexities running on complex modern general purpose processors accurately for single core and multi-core AMP mode scenarios, such that the method is simple to use, generic and most importantly open-source?'.

1.2 Motivation

Digital signal processors are a natural choice for audio processing and assembly programming is the preferred choice. Several features of DSP's make them favourable for audio signal processing. The most significant of these features are:

- VLIW architecture - DSP's possess Very Large Instruction Word architecture where multiple inctructions can be grouped together to form one large instruction such that instructions can execute concurrently in multiple execution units.
- Parallel buses - DSP's have *multiple* program and data buses which enable parallel movement of data.
- Specialized addressing modes - DSP's support modulo and bit-reversed addressing of data which directly benefit algorithms such as FFT (Fast Fourier Transform) which need to perform bit-reversal.
- Less dynamic features - Absence of dynamic features such as advanced branch prediction, caches etc.. help in realising real-time constraints.

Despite the above favourable features, DSPs are not evolving at the same rate as general purpose processors and programming in assembly is rather tedious. Also, high-end DSPs are more expensive than high-end general purpose processors although they run at lower frequencies compared to the latter. On the other hand, general purpose processors are evolving at enormous speeds with increasing frequencies and number of cores. Choosing a multi-core software design is soon going to be inevitable as algorithm complexity increases and single core processors hit the power wall.

Can complex audio algorithms written in C programming language executed on general purpose processors running at higher clock frequencies provide the same performance and timing guarantees as DSPs? Can homogenous general purpose multi-core processors running in AMP mode, with control on one core and audio algorithms running on another core either bare-metal or with a RTOS if its multithreaded, still adhere to real-time deadlines? Can multiple algorithms be ported on the same general purpose processor core and still adhere to deadlines? These questions form the motivation for the problem statement. The next section describes the approach taken to solve the problem presented in the problem statement and key contributions of this thesis.

1.3 Approach and thesis contributions

As described in section 1.1, a method to determine execution time of audio algorithms on single core and multi-core modern general purpose processors was required to be developed. The method we chose to develop was a flexible model/simulation framework to be able to adapt to the dynamic nature (run-time inter-core communication in AMP mode) and complexity (complex general purpose processors) of the problem.

In order to achieve this, a top-down approach was employed. We evaluated execution time determination at different abstraction levels namely - Algorithmic block, basic block and instruction level using various simulators. The initial focus was on program execution time accuracy of audio algorithms running *bare-metal* on single core. The multi-core scenario was considered at the end. Functional simulation or hardware emulation without timing was not considered in this thesis. Simple audio algorithms were used for the various experiments and the results were compared with hardware measurements. The general purpose processor chosen was the cortex-A9 housed in a Zedboard SoC belonging to the Zynq family of Xilinx SoCs. The results of various abstraction levels and methods were compared and eliminated based on accuracy, flexibility, hardware compatibility and scalability.

The main contribution of this thesis is experimental evaluation of execution time determination at various abstraction levels using various simulators and suggestion of a suitable simulation framework to conduct execution time analysis of audio algorithms for single core and multi-core AMP mode scenarios.

1.4 Thesis outline

In this chapter we describe the problem statement, approach taken and thesis contributions. The rest of the thesis is organised as follows: Chapter 2 provides the background on audio signal processing, modern general purpose processors and execution time analysis. Chapter 3 explains the related work in the domain of execution time estimation categorised under different abstraction levels. Chapter 4 describes the first experiment done at the algorithm block abstraction level for execution time analysis. The chapter also discusses the results and trade-offs. Chapter 5 describes experiments at the basic block abstraction level. A basic block is a block of instructions with one entry and one exit point. Chapter 6 concludes the experimental section describing the experiment at cycle-accurate instruction level abstraction. Chapter 7 concludes the thesis with descriptions of the simulation framework suggested, future work and general conclusions regarding execution time analysis of audio algorithms on modern GPPs (general purpose processors).

2 Background

2.1 Audio signal processing

The sound we hear around us can be classified under a kinetic form of energy called Acoustical energy. Acoustic energy comprises of fluctuating waves of pressure in the form of compressions and rarefactions of air molecules as they propagate along air.

An audio signal is an electrical representation of a sound, in the form of a fluctuating voltage or current. Depending on the A/D convertor resolution and limits of audio equipment, the signal fluctuates at the same rate as the acoustical energy that it represents, and the amplitudes of the acoustical sound wave and the electrical audio signal are scaled proportionately [9]. The analog sound inputs are hence digitised to produce discrete-time audio signals.

2.1.1 Signal flow

Audio signals may belong to speech or music. The audio signals are processed to alter one or more characteristics of the audio signal. The processing is usually done by digital signal processors which are used for discrete-time signal processing. The hardware architecture and instruction set architecture of DSPs are tuned for real-time signal processing algorithms.

Figure 2.1 shows the signal flow in audio applications. I2S stands for Inter-IC sound which is a protocol for streaming audio samples. The incoming audio from an analog source passes through an A/D convertor which samples the analog data at a rate equal to twice the fundamental frequency of the analog input. This is also known as the nyquist sampling criterion. I2S splits the digitised samples alternatively into right and left channel audio data. This is a desirable property for most audio applications requiring stereo sound output (right and left speakers). If mono sound (one channel) output is desired, samples from one channel can be discarded.

Most audio algorithms work with blocks of audio samples instead of individual samples which requires buffering of digitised audio samples. Working with blocks of samples is also beneficial from a processor performance point of view as the core does not have to process individual sample interrupts. Thus, the right and left channel audio data gets buffered in the I2S FIFO. DMA (Direct memory access) is configured to transfer 'N' samples (N is the block size) from the FIFO to either DDR (Double Data Rate) memory or internal memory and then sends an interrupt to the core, which then processes the samples. From the Figure 2.1 it is clear that the deadline for audio processing to complete

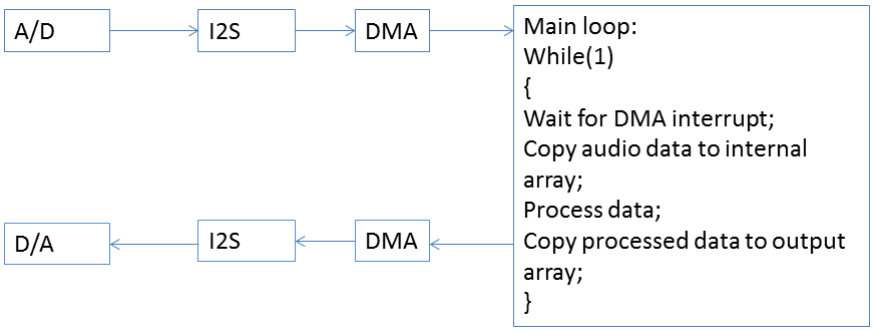


Figure 2.1: Signal flow in audio applications

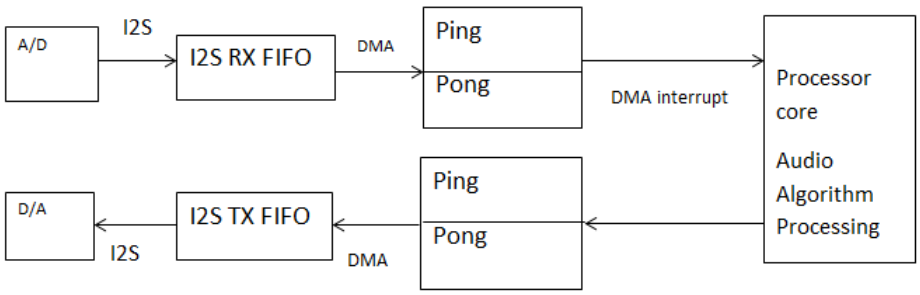


Figure 2.2: Ping-pong buffer scheme

is before the next DMA interrupt is triggered. The return signal flow path is similar to the forward path.

In order to ensure that no audio data is lost because of overwriting the data in the buffers while the core is still processing the audio samples, a ping-pong buffer scheme is adopted in audio applications. Figure 2.2 shows the signal flow with the ping-pong buffers in place. The idea is that while DMA is using the ping buffer, the processor is using the pong buffer. When they are finished manipulating the current block of samples, the processor and DMA will exchange the buffers they were using earlier. This holds good for both the forward and return paths. An interrupt is raised at the point of exchange. On the RX side, DMA transfers incoming audio data to ping buffer while the core is manipulating previous block of samples in pong buffer. On completion of transfer, DMA triggers an interrupt to signal the core that new data is available after which the buffers are exchanged. Similarly, on the TX side, the processor triggers an interrupt to the DMA once it completes copying the current output data to pong buffer while DMA was transmitting the previous output samples in ping buffer after which the exchange occurs.

2.1.2 Requirements of real-time audio applications

Audio applications running on DSPs or modern general purpose processors must satisfy certain requirements as follows:

1. **End-to-end latency** - This is the latency from the arrival of the first incoming audio sample at A/D end before conversion to the first outgoing audio sample at the D/A end after conversion. This observed latency must be within a permissible limit to satisfy certain quality of service standards.

2. **Functionality** - Along with audio algorithm processing, audio applications need to cater to several other functionality such as communication of control commands, keypad scanning, display control, analog to digital conversions and vice-versa. These functionalities need to be met as well.

3. **Throughput and response time** - As audio applications become more complex with the above mentioned functionalities, throughput and response time of the application become more crucial especially if the application has hard real-time response requirements to adhere to certain quality standards.

4. **Memory usage** - The application is restricted to use only the memory available in the underlying hardware.

5. **Scheduler** - If the application is multithreaded, an Operating system needs to be used. For bare-metal applications, the scheduling is mainly infinite main-loop based such as while(1) loops.

6. **Deadlock prevention** - If synchronization primitives such as semaphores and mutexes are used to protect shared resources in an audio application employing an OS, the system can easily run into deadlocks. Thus, deadlock prevention is a very important requirement for an audio application using an OS.

An elaborate performance analysis is required to determine if the above requirements are satisfied for successful operation of the audio applications.

2.2 Modern Processors

Modern processors are evolving in terms of performance and complexity at an enormous speed. The advances in technology from simple microcontrollers to modern processors have been exponential. Since our thesis focusses on ARM processors, we will discuss internals of modern ARM processors in this section. ARM, a popular low-power IP supplier, was founded in Cambridge, UK in 1990. They supply a wide range of microcontrollers and microprocessor IPs targeted at various applications. Here is a peek into the modern ARM processor internals:

1. **Deep pipelines** - Simple processors like ARM8 consist of 5-stage pipelines. More recent processors like the cortex A15 consists of a variable length pipeline varying between 17 - 25 stages. Most high end processors have a front end and a back-end pipeline. The front end consists of stages like Fetch, Decode, Rename and Dispatch. The back-end consists of Issue, Execute and Writeback. Deeper pipelines introduce more complexity in terms of pipeline stalls and pipeline flushes which have to be monitored for execution time analysis.

2. **Speculation and prediction** - Modern processors consist of complex branch prediction schemes and branch target address caches to hold addresses of branches predicted as taken. This enables speculative execution of instructions from branch addresses as indicated by the branch target caches. If the branch is indeed taken the results of speculative instructions are committed else the pipeline is flushed and the instruction at the new resolved branch address is fetched.

3. **Dynamic scheduling** - Modern processors perform instruction scheduling in hardware which enables them to schedule and issue instructions out-of-order into the multiple execution pipelines. Register renaming prevents Write after Write (WAW) and Write after Read (WAR) stalls. Hence instructions without any true data dependencies i.e RAW (Read After Write) stalls can be issued out-of-order.

4. **Caches** - Caches contain copies of limited amount of main memory enabling fast access. Modern processors have multiple levels of cache hierarchies with complex allocation and replacement policies which make cache accesses highly non-deterministic. Furthermore, caches are non-blocking and can handle outstanding misses without stalling the processor. Also, cache lines can be prefetched from main memory if cache misses can be detected earlier.

5. **MMU** - Virtual memory management was not present in the earlier microcontrollers but its a common feature of modern microprocessors. The application running on the processor is mapped onto virtual memory space and page tables containing physical addresses have to be read to obtain the physical addresses. This is known as page-table walks. The page-tables are located in main memory and the translations get cached in small caches called as translation look-aside buffers. TLB misses prove quite expensive as page-tables in the main memory have to be read which takes large amount of cycles.

6. **Multicore cpus** - Multi-core cpus are fast gaining popularity in the modern processor category which brings cache coherence and shared-memory conflicts into the picture. Thus cache coherence protocols and synchronization primitives such as semaphores between processors have to be introduced which add to the non-determinism.

The evolution of complex SoCs and NoCs take the complexity a level higher by integrating several other IPs with the processor to form a SoC or a NoC. Several interconnects are used with different protocols to connect the processor with off chip peripherals and memory controllers. Thus, each component on the SoC has to be examined closely as it

further adds to the latency of the source program running on the processor depending on which peripheral/component is utilised by the software. Hence modern processors on complex SoCs pose a big challenge for deterministic execution time analysis.

2.3 Execution time analysis

Execution time of a program depends both on the software behaviour (code structure) and the hardware (hardware timing) on which it is executed. Software development cycle for real-time embedded systems typically includes execution time analysis of the source program.

2.3.1 Need for timing analysis

The analysis can be conducted for the following reasons:

- Worst case execution time determination - For hard-real time systems its desirable to know the worst case execution times of tasks or time-critical code fragments to be able to analyse if they will finish executing within the deadline. Hence determining the task execution time is paramount for real-time systems.

- Tracking deadline misses - Audio algorithms have hard real-time deadlines. These deadlines are usually the sampling period of individual samples or blocks of samples. The audio processing on the current sample must be finished before the next audio sample becomes available in the A/D buffer. Failure of processing within the individual/block-sample period results in a deadline miss and the audio sample gets overwritten in the A/D buffer. This effect is somewhat mitigated by using ping pong buffers as explained in Figure 2.2. Tracking deadline misses becomes very important to detect loss of audio samples and how it affects the quality of service.

- Throughput and latency determination - For audio applications, latency is a more important metric than throughput as its a hard real-time system. The end-to-end latency of the system employing audio algorithms must satisfy deadline constraints. For video applications, throughput would be a more important criteria as its a soft real-time system and the number of output frames per second should be high for high quality.

- Performance optimization - Software developers developing the algorithm in C language typically write code which is not optimized for any specific hardware. Execution time analysis of the program running on the intended hardware exposes possible performance optimizations which can be performed such as prefetching to reduce cache misses or using SIMD (Single input multiple data) co-processors available on some processors. This becomes critical if deadline misses are detected during execution time analysis.

- Processor utilization - Execution time analysis is also important to know the percentage of processor utilization in order to determine how many more processor cycles are left to do other tasks or run additional algorithms.

2.3.2 Traditional methods of execution time estimation

For audio applications, initial algorithm development is typically done in Matlab for functional verification before the C code is written. The C code is then ported onto the processor and profiled for analysis, for reasons mentioned in subsection 2.3.1. Traditionally, the following methods have been adopted for execution time analysis of source programs:

1. **Measurements using LEDS** - Turning an LED on and off before and after the code segment whose execution time needs to be measured. The duration of the pulse width corresponds to the execution time.

2. **Timers** - High resolution timers available on the processor can be used to measure task/code segment's execution time. Typically, the timers are loaded with the highest possible value and started just before the code segment starts executing. The timer starts counting down, hence reading the value upon execution of the code segment and subtracting it from the initial high value yields the execution time of the code segment.

3. **Logic Analysers/Oscilloscope** - These are tools which aid in observing the LED pulses for execution time determination or even observe instruction and data transactions on the system bus and external pins. These tools are often used to debug hardware behaviour but can also provide visibility of above mentioned factors such as LED pulses, program and data bus transactions which can help in execution time analysis.

4. **Time stamps** - System calls such as gettimeofday() in Linux can be used to get time stamps before the start and after the execution of the code segment if an OS is being used. Resolution upto microsecond can be obtained with these system calls.

5. **Static analysis** - Static analysis refers to calculating execution time using static means without executing the source code. The processor manual lists instruction cycle timings and knowing information about the processor pipeline, one can calculate the execution time of the source code. Static analysis is typically useful to determine worst case execution time of embedded software where input-dependent program flow variability exists such as interrupts, incoming data in streaming applications etc.

Profiling is a very common technique to gain insight on program performance and behaviour. It is an intrusive method where the source code is instrumented for profiling

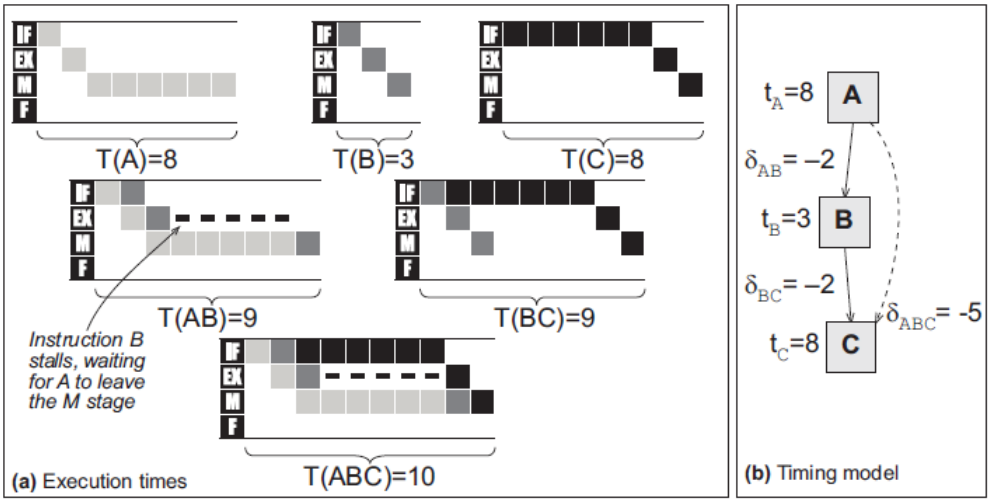


Figure 2.3: Example of long timing effects. Source: [1]

by the compiler (in most cases) and the program counter is sampled after every periodic interval. Through this way the instruction and hence the function being executed is determined. Main aim of profiling applications is to determine in which function the program is spending the most amount of time. Profiling of applications is typically done **after** porting an application on the hardware. Popular profilers are gprof from GNU cross compiler and Valgrind (refer subsection 3.4.2) which also gives information about cache misses and branch mispredictions.

2.3.3 Timing Abnormalities

Modern processors exhibit timing anomalies which makes execution time analysis and hence performance modelling even more complicated. Furthermore, the timing anomalies are very dynamic in nature as it depends on internal processor and pipeline states and may not show up for a particular measurement. This section discusses various timing anomalies and their sources.

2.3.3.1 Long timing effects

A long timing effect for a sequence of instructions $I_1 \ldots I_m$, $m \geq 3$, occurs whenever I_1 has the effect of disturbing the execution in such a way that the execution of the instructions $I_2 \ldots I_m$ is different compared to if I_1 had not been present. This occurs precisely when I_1 causes a stall to some instruction following it[10].The long timing effects can be negative or positive and can occur over variable instruction windows. Thus, analysis methods employing analysis over pairwise instructions or short instruction windows might not be able to estimate long timing effects on the execution time. The example in figure 2.3 shows an interaction between instructions A and C that is not visible when just analyzing the neighbouring pairs of instructions AB and BC [1].

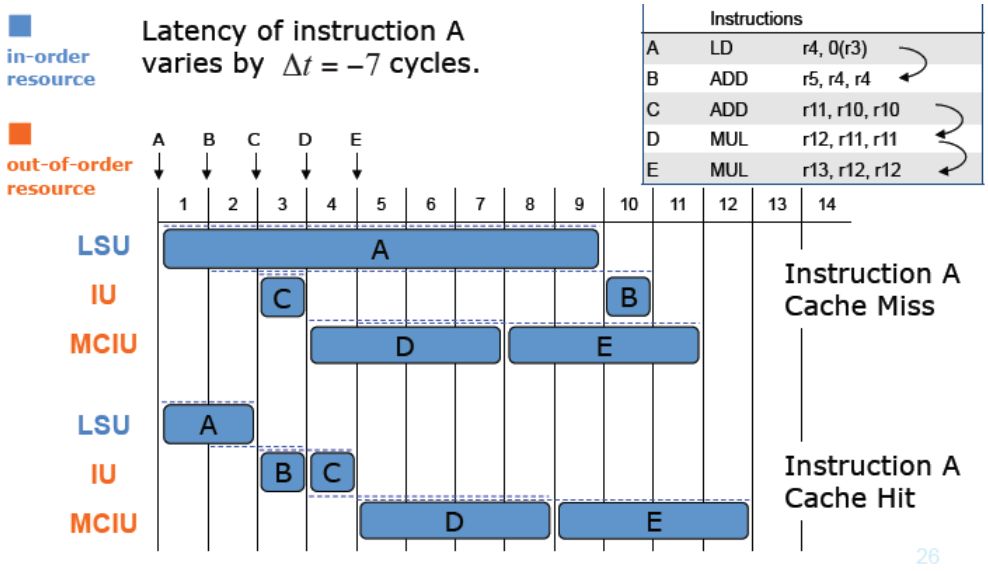


Figure 2.4: Example of Cache timing anomaly. Source: [2]

2.3.3.2 Cache anomaly

Caches in modern processors add dynamism and unpredictability to the problem of execution time analysis. Cache features such as associativity, sizes, cache policies and cache replacement policies influence execution time. The underlying memory system along with caches make execution time analysis for memory-intensive and communication-intensive algorithms more complex. Furthermore, out-of-order execution interspersed with in-order resources interacting with complex caches and memory systems expose certain timing anomalies introducing variability in the execution time. Figure 2.4 illustrates an anomaly when cache hits can take more cpu cycles than cache misses for the instruction mix mentioned. These anomalies are very dynamic in nature and depend on the instruction mix the code structure introduces and also on the interaction between pipeline resources and the multi-level memory systems in modern processors.

2.3.4 Summary

Execution time analysis is paramount in determining system performance to assure quality of service for hard real-time systems and for several other reasons as mentioned in subsection 2.3.1. Traditional methods of execution time like the ones mentioned in subsection 2.3.2 fail to capture complex and dynamic effects like the ones discussed in subsection 2.3.3 and hence can be very inaccurate. Profiling methods are more intrusive and are not meant for pure and accurate execution time estimation.

Execution time estimation at different abstraction levels

3

3.1 Introduction

An abstraction is a method of specifying properties of certain system behaviour relevant to the job at hand sufficiently, abstracting out irrelevant specific details. Choosing the appropriate abstraction level is paramount in performance analysis. Hence models of embedded systems analysed for execution time estimation can be roughly grouped into three categories of abstraction levels namely:

- Loosely timed or functional models.
- Approximately timed.
- Cycle accurate.

Before explaining the abstraction levels further, let us try to answer:

3.1.1 What is a model?

A model is an approximate representation of a system, which is intended for the analysis of certain properties.

3.1.2 What are the advantages of modelling?

- Models allow the analysis of functional and non-functional properties before actually realizing the system in hardware and software.
- Modelling aids system specification (application and hardware platform) in a more abstract manner abstracting from the more obvious details of the platform or detailed details of the application.
- Modelling the system and conducting a performance analysis of it helps in confirming design choices such as the hardware platform.
- To investigate the feasibility of design alternatives without actually realising them, models can be used.
- Modelling a system significantly reduces time to market as performance analysis leads to identifying potential bottlenecks and hence changes can be made at the prototype stage to make the transition to final product smoother.

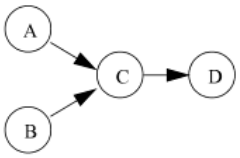


Figure 3.1: Data Flow Graph. Source: [3]

- When expectations are unclear and performance analysis with an approximation is desired, models can be very helpful.

3.1.3 Challenge of modeling

Abstraction focusses on disregarding irrelevant details, whereas adequacy requires including all relevant aspects. These conflicting objectives of abstraction and adequacy make construction of appropriate models to meet expected goals difficult.

3.2 High-level modelling techniques

3.2.1 SDF (Synchronous Data Flow)

Dataflow is a well-known programming model in which a program is represented as a set of tasks with data precedences. Figure 3.1 represents a dataflow graph, where computation tasks (actors) A, B, C, D are represented as circles and arrows (or arcs) between these actors represent FIFO (first-in first-out) queues that direct data values from the output of one computation to the input of another. Actors consume data (or tokens represented as bullets in Figure 3.2) from their inputs, perform computations on them (fire), and produce a certain number of tokens on their outputs [3].

Each actor has a specified rate of consumption and production of tokens. Synchronous Data Flow graphs are those for which the rate of production or consumption of tokens is known a priori. That is, the consumption or production rate of tokens is independent of incoming data. Most signal processing applications possess this synchrony, hence SDFs are useful in specifying and analysing DSP and streaming applications[11].

In [12] it has been shown how SDFs can be used to perform timing analysis of streaming applications such as JPEG decoder running on a multiprocessor NoC. A more specific form of SDF called homogenous SDF (HSDF) models are employed in this paper. HSDF follows a simple firing rule: every actor consumes one token and produces one token upon firing. The job flow in the application is specified using HSDF graphs called computation graphs. The actors in the computation graphs represent processor code segments executed on one processor with execution time annotated within them. The edges between them show either data dependencies or execution order. The intra-job scheduling

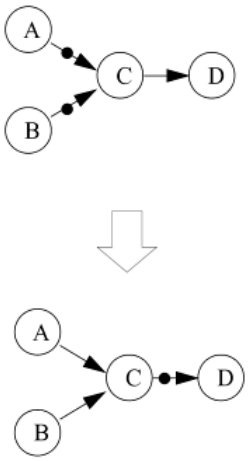


Figure 3.2: Actor firing. Source: [3]

follows a static cyclic order dictated by the order of firings of the actors. The connection between these processor nodes are implemented by HSDF components called channels, which are the basic communication primitives for jobs in a multiprocessor NoC context. A channel consists of input buffers, a data connection, a flow-control connection and output buffers. All the HSDF components are assembled together to form one inter-process communication graph. The paper focuses on deriving inter-process communication (IPC) graphs from computation graphs and configuration networks comprising of channels and determining worst case behaviour by applying timing properties of IPC graphs.

Worst case execution time of actors for the JPEG decoder case study have been determined for ARM7TDMI processor by using an Instruction Set Simulator (ISS) from ARM ltd. Single-cycle memory access with no caches have been assumed. The real execution times have been compared to the execution time derived by applying the theorems mentioned in the paper for the IPC graph of the JPEG decoder. It was observed that feeding an upper bound for actor execution time determined by evaluating architecture-dependent parametric expressions yields only 20 % overestimation compared to feeding absolute worst-case execution times for any input image. Another significant contribution of the paper was determining rate-optimal buffer sizing by varying buffer sizes and performing critical cycle analysis (the cycle of the graph yielding maximum execution time) of the IPC graph.

The two obvious disadvantages of the above performance analysis method are: the scheduling of various actors is determined by a static order and fixed worst case execution times of actors have to be fed to the IPC graph. This problem has been addressed in [13] where a novel SDF called scenario-aware data flow (SADF) model has been proposed which allows the scenario to influence the execution time of the actors. It also takes into account that actors can have zero data dependencies in certain scenarios and hence allows zero production and consumption rates in those scenarios. On the other hand, the amount of data produced may vary with different scenarios and this can be specified

in the model as well. The approach has been applied to a MPEG-4 decoder case study and its performance is analysed for different type of input frames which correspond to different scenarios.

SADF consists of kernel actors which perform data processing and detector actors which model the control flow. The detectors have underlying Markov chains associated with them which handle state transitions of the detector. The transitions are based on occurrences of scenarios according to a timed probabilistic labelled transition system. Kernels have a control port attached to detectors and the token on this port determines the scenario for the kernel operation.

The execution time distribution of actors and probability distribution of scenario occurrences are determined using a profiling tool. A huge set of all possible input frames for the application are considered during profiling to confirm the probability distribution. Details of the profiling tool and the underlying processor are not mentioned. Simulation based performance estimation using POOSL (refer subsection 3.2.2) is used in the paper. The SADF is specified using XML notation and a tool called sadf2poosl automatically translates SADF specification into a POOSL model which is simulated using rotalumis, a simulator for POOSL models. The paper reports accurate throughput results for certain kernels which is equal to expected results (reciprocal of average execution time between successive firings which is also calculated from the graph). Large variance in successive firings for few kernels is reported indicating arrival of tokens in bursts. Other performance metrics such as average occupancy and maximum occupancy of buffers have also been simulated and analysed. Thus a method for performance analysis of streaming applications with dynamic inputs has been demonstrated in [13].

3.2.2 POOSL modelling language

Software/hardware engineering(SHE) is a general system-level design methodology which uses POOSL as the formal modelling language and UML as the informal modelling language.

POOSL stands for Parallel object-oriented specification language. It is a specification language through which your application and platform can be specified for analysis. UML stands for unified modelling language which is a framework for describing models for systems using graphical or textual notations.

3.2.2.1 Features of SHE methodology [4]

- System behaviour - SHE enables understanding hardware-software systems very well by allowing us to create very intuitive and easy to comprehend models using POOSL.

- Extensions to UML - SHE is developed upon extension of UML to enable scheduling, performance and specifying time to develop abstract models after describing requirements of the system.

- Mathematical analysis techniques - SHE adds formal (i.e mathematically defined) analysis techniques to UML which are required for expressing complex embedded system behaviour and analysing performance properties.

- Executable models - SHE enables transformation of models developed in UML and specified with POOSL to executable models which is required for performance analysis. SHEsim is a tool which can simulate these executable models and thus be able to do time-accurate performance analysis.

- Evaluation of requirements - SHE allows us to specify functional (intended behaviour) or non-functional requirements (performance requirements) of the system. These are evaluated once the task and resource mapping is completed and based on the evaluation results , the task model or the resource model or the mapping can be tuned until the evaluation is successful.

- Formal semantics - Proper application of mathematical analysis techniques is supported by the formal semantics of POOSL. The formal semantics of POOSL combines ideas of traditional imperative object oriented programming languages with a probabilistic real-time version of the process algebra. Modelling languages like POOSL based on formal semantics result in unambiguous performance metrics.

- Expressive Power - SHE has incorporated several extensions to UML to make POOSL more intuitive and expressive. POOSL consists of several primitives and library of classes to adequately express the behavior of the hardware/software system.

- Object-oriented - POOSL has three main constructs:

  1. Processes
  2. Data objects
  3. Clusters.

  Processes are tasks which are capable of independently executing and are defined using statements which in turn consist of data objects and methods. All processes are executed in parallel. Processes are able to communicate with each other by sending messages across channels. Clusters are aggregation of processes and other clusters. POOSL offers several primitives and statements to adequately describe the hardware/software system using the above classes. The POOSL model is then converted to an executable which is simulated by a simulator called SHEsim. SHEsim is also a system level editor which provides a GUI interface for defining the models using the above mentioned POOSL classes as shown in Figure 3.3, and also running the simulation.

- Probabilistic data - SHE also allows for expressing probabilistic data which might be important for performance evaluation.

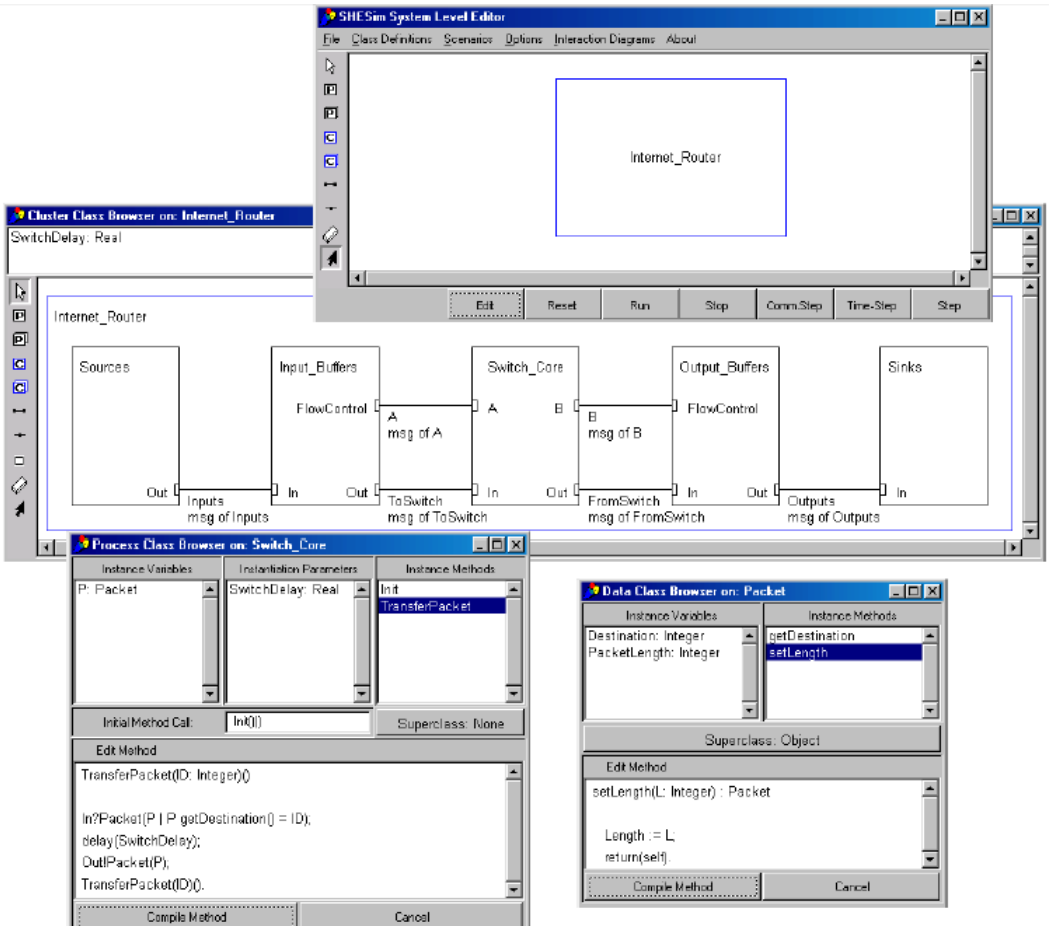


Figure 3.3: A snapshot of GUI showing different POOSL classes. Source: [4]

3.2.2.2 Conclusion

SDF graphs are most popular modelling approaches to conduct high-level performance analysis on complex streaming applications. POOSL is a very expressive modelling language used to **specify** the application and hardware platform and conduct performance analysis by simulating the executable model.

3.3 Basic block methodology

The basic block is the most popular 'unit' of measurement in the abstract modelling domain. A program can be considered to consist of several basic blocks. It is a block of instructions at the assembly level or the source code level with just one entry and exit point for the control flow.

3.3.1 Static methods

3.3.1.1 Mathematical model

Mathematical models to determine execution time of a source program usually consist of two parts: pure computation time and dynamic information such as cache misses and branch mispredictions multiplied with their respective miss penalties. If the instruction mix throughout the program is similar then the CPI of a basic block can be statically calculated or measured and multiplied with the dynamic instruction count of the whole application. Measured CPI metric rather than statically determined, averages out timing differences due to processor pipelines, superscalar execution etc.. as mentioned in[14]. Another method of determining pure computation time is to multiply individual instructions with their execution times and loop bounds if any, which obviously is infeasible for large programs. Dynamic cache misses can be determined by considering the context sizes (number of variables * size of the type), cache size boundaries and cache policies (write back, write allocate etc). This multiplied with the cache miss penalties gives the total communication cost which added to the pure computational cost gives us the total execution time. The mathematical model presented in [14] does not consider branch prediction schemes, branch mispredictions, pipelining stalls due to hazards and concurrent pipelines in modern processors. Mathematical models based on basic blocks like the one presented in [14] works if the whole application has basic blocks containing similar instruction mix as the one for which the CPI is computed. This holds true also for the assumption made that the CPI metric measured by executing the basic block on hardware averages out timing differences due to features of modern processors. Furthermore, modern processors consist of non-blocking caches. Thus, the caches are capable of handling certain number of outstanding misses and the processor is not stalled on a cache read/write miss. The instruction which caused the stall is re-issued once the data is in the cache. These caches cleverly hide memory latencies, the effects of which cannot be analysed easily.

Since mathematical models have certain limitations as mentioned above they were not explored in this thesis.

3.3.2 Hybrid methods

3.3.2.1 WCET

Hard real-time systems often require analysis of worst case execution time of the tasks. A hybrid approach involving static analysis of the compiled object code of the application program and measurements using an ISS (Instruction Set Simulator) is employed in [10] to determine WCET. A tool architecture which splits the WCET analysis into several modules for higher flexibility and easy testing is proposed by the author. The tool chain aims at broad applicability of the analysis for various different processors . This paper has applied the pipeline analysis and performed experiments with their WCET prototype tool on ARM9 and NEC V850E processors. The WCET analysis modules proposed by

the author are flow analysis (to determine possible program flows amongst basic blocks), global low-level analysis (to consider cache effects across basic block boundaries), local low-level analysis(to consider pipeline effects) and calculation of the WCET .Timing analysis is performed at the basic block abstraction level and it combines results of CPU simulators, global and local low-level analysis to determine concrete execution times. Hence the timing analysis consists of a timing graph where each node representing a basic block is accompanied by an execution scenario. The execution scenario considers timing effects due to pipeline overlap or pipeline stalls while execution of basic blocks and also cache misses. The final WCET calculation is determined using IPET (Implicit path enumeration technique) which uses both the flows and execution times from the timing model.

The paper shows 20% reduction in overestimation by using extensive pipeline analysis. The paper presents a formal mathematical model of in-order processor pipelines as a set of equations representing constraints imposed on them. These constraints express correlation between instructions and pipeline stages for branch instructions, data dependence, structural hazards and superscalar pipelines. Manual annotations consisting of facts such as maximum loop bounds have to be specified for deriving scope graphs which are in turn used for constructing the timing graph. In static WCET analysis, a branch is always predicted to be taken as this leads to longer execution times than the fall-through case. A cpu simulator/emulator has been used to provide execution times of basic blocks for worst-case traces for the NEC V850E processor. Tables with execution times provided in ARM manuals were used as ARM9 hardware model. Cache analysis is not dealt with in the experiments as none of the target architectures use caches.

Thus, the thesis explains a hybrid WCET approach where execution times of instruction traces given by a simulator is subjected to detailed static pipeline analysis. However, the prototype tool is not publicly accessible and aspects such as complex out-of-order pipelines have not been modelled. The execution scenario of each basic block node does not consider dynamic aspects such as data cache accesses and branch prediction making the approach presented in this thesis somewhat difficult to implement considering cortex-A processors.

However, since the paper reports high accuracies for execution times for known worst cases, a WCET tool with similar modules of flow construction, cache and pipeline analysis called aiT which is available under a free academic license has been explored.

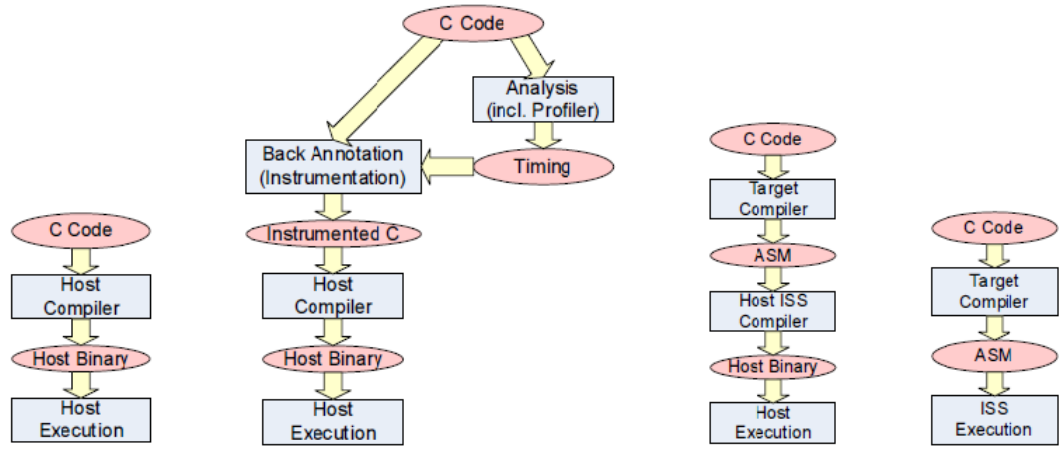
3.3.2.2 System-C TLM

Transaction level modelling (TLM) is an IEEE standard and provides an interface for modelling of buses, interconnects and memory accesses. TLM is accepted widely as a technique for efficient abstract modelling of communication in system level modelling.[5] proposes a technique to apply TLM concepts for computation. This is based on the premise that both communication and computation share the same concepts of functionality and timing. Since TLM is mainly based on the separation of concerns of functionality and timing it can be equally applicable to both computation and communication. The

biggest contribution of this paper for our thesis is the emphasis on abstract modelling, separation of concerns and result oriented modelling. Optimistic execution times are calculated for a process till the next observable activity in the system. Disturbing influences which occur during the runtime of the process can change system state. The optimistic execution times of the process are then corrected at the end of the process due to these disturbing influences. The paper suggests that pipelines and caches in modern processors can be modelled as disturbances. The goal of result oriented modelling (ROM) is to produce end results of a process at a higher simulation speed eliminating analysis of intermediate internal states in the process. The ROM approach has been applied to bus transactions between two masters and slaves on the AMBA AHB bus and compared with TLM and Bus functional models for performance in terms of simulation bandwidth and timing accuracy in terms of transfer of duration (compared to the standard). Disturbances such as a higher priority bus initiating transactions have been analysed. The paper reports 100% accuracy when compared to the accurate bus functional models.

[15]focuses on improving the timing accuracy of TLM computation models used in system modelling. The author proposes a timing annotation methodology using basic blocks as estimation units. The basic blocks are determined from the object code and not the source program. The paper explains why basic blocks are suitable estimation units for timing annotation. Since basic blocks contain just one entry and exit point so essentially one execution sequence, it is suitable to annotate the TLM computation model with a single delay value per basic block. Also, the basic block contains suitable number of instructions to perform accurate hardware pipeline and cache analysis. Raising the abstraction level to basic block level compared to instruction level as done in Instruction Set Simulator (ISS) improves the simulation speed. The timing annotation flow consists of developing a control flow graph (derived from the source program) and performing timing estimation of the basic blocks (derived from object code). The basic block cycle count is done statically by tracking pipeline status and considering stalls due to data dependence. This is combined with boundary effect correction across successive basic blocks due to pipeline overlapping or pipeline saturation due to superscalar (parallel) execution. Dynamic information such as branch prediction is implemented somewhat primitively using a predict function which has two functions: to return a boundary correction factor if the current block is the predicted successor of the preceding block and to predict which basic block will be implemented next. Cache access time adjustments based on a Icache model are included in the basic block time estimation. Finally, a TLM computation model written in system C is generated which simulates the control flow as per the control flow graph and is annotated with basic block time estimates using delay statements. It also includes predict function which functions as mentioned above.

The paper reports error rates within 2% of those estimated by a cycle-accurate ISS for program execution times. The selected target architecture is SimpleScalar PIZA [16]. A similar approach as the time annotated basic block methodology is adopted in [17] which generates a back-annotated system C model which is simulated for execution time estimation. However, the author also proposes a branch prediction model which implements a state machine consisting of taken and not-taken cases depending on the prediction scheme incorporated in the processor. Note: basic block composition is such



(a) Untimed Spec (b) Computation TLM (c) Host Compiled ISS (d) ISS

Figure 3.4: Comparison of different abstract levels for model of computation. Source: [5]

that branches can occur only at the end of each basic block which makes dynamic correction due to branch prediction simpler. Depending on the executed branch behaviour at the end of the basic block, the correction cycle count for predicted or mispredicted branches is added to the static cycle count of the basic block. The author also uses a processor description model which specifies the pipeline stages and functional units utilised by each instruction in the instruction set. This processor description aids analysing data dependence within the basic block during static calculation of basic block cycle count. The error rates reported in [17] are between 4 and 7% when compared to actual hardware but simulation speeds 91% times higher compared to Instruction Set Simulators. The target is a Tricore processor.

Since addresses for data accesses are sometimes mentioned using indirect addresses (values in register contents) which can be determined only by running the binary, data cache misses are hard to detect. Therefore average data access times are considered in the above methodology sacrificing some accuracy. Also, cache-level hierarchy has not been considered. On the other hand, the ease of specifying functionality and timing separately in a system level description language like systemC is what is taken advantage of in this methodology. The simulation time proceeds only when wait for statements are executed, hence the code corresponding to functionality is executed in delta cycles contributing nothing to simulation time. Furthermore, raising the abstraction to basic blocks leads to fewer wait for statements leading to faster simulation as costly context switches are not performed in the simulator because of wait for statements. Due to several obvious advantages, basic block methodology has been explored in this thesis.

3.4 Instruction level functional simulation (Loosely timed models)

Functional simulation refers to simulating/emulating the underlying hardware to produce the desired 'functionality' of the application as though it was executing on the emulated hardware. As stated in [18] "To accomplish this effect, an ISA emulator needs to keep track of the guest program state, and dynamically update it instruction by instruction until the program finishes. The state of a program can be most generally expressed as its virtual memory image and the architected register file. The virtual memory image consists of the set of values stored at each possible memory location addressable by the program. The state of the architected register file is formed of the values for each register defined in a specific architecture (ISA)".

3.4.1 Dynamic binary translation (DBT) - QEMU

Qemu is a hardware emulator based on dynamic binary translation. For instance, if the processor to be emulated is Arm and Qemu is running on an X86 processor, it translates Arm executable into X86 executable during runtime. It performs a basic fetch-decode-execute loop. Qemu has to be used with a cache simulator such as Dinero [19] and a custom branch predictor has to be written. It does not model time, its purely a functional hardware emulator. Therefore the time has to be calculated based on graduated instructions and other data derived from additional simulators as done in[20].

$$cycles = \frac{(Ig * L1_{ht})}{i_{fs}} + DL1_a L1_{ht} + L1_m L1_{mt} + L2_m L2_{mt} + Br_m Br_{mt} \tag{3.1}$$

where I_g is graduated instructions which are nothing but instructions which have been successfully 'written back' to completion. $L1_{ht}$ is the L1 cache hit time, i_{fs} is the instruction fetch size(4 words), $DL1_a$ is L1 data cache accesses, $L1_m$ is L1 misses and $L1_{mt}$ is L1 miss time. Br_m is branch misses and Br_{mt} is branch miss penalty.

3.4.2 Dynamic binary instrumentation (DBI) - Valgrind

Valgrind provides a framework for program analysis and profiling based on dynamic binary instrumentation.

The tool chosen by the user (Cachegrind, memcheck) injects some arbitrary code in the binary to be analysed and this is run on a synthetic cpu running the target processor instruction set. It is mainly used as a memcheck tool to check for memory leaks but it also provides a tool called cachegrind which gives information about cache misses . Branch prediction can be simulated as well and a branch predictor similar to the ones found on x86 machines is simulated.

3.4.3 Fast Processor models - OVP

OVP models are functional models and instruction accurate but do not model pipeline or architectural features such as superscalar issue and out-of-order execution. The models do not model processor speed and instead have mips rating. The number of instructions executed in a measured elapsed amount of time called a quantum are calculated according to the formula: Instructions per quantum = mips rating * quantum where quantum is configurable by the user.

Provides performance monitors as a register interface only. The model translates target processor code to native code using certain APIs [21] and the OVP simulator performs just in time compilation for native code execution. OVP models need a simulator called OVPsim to simulate the models. The OVP models can also be integrated into system C.

3.5 Cycle accurate methods

3.5.1 Hardware measurements

Modern processors come equipped with performance counters which can be used to gather useful information such as cycle count of CPUs. Cortex A9 processors provide performance monitoring units for each core of the processor cluster. The PMU provides 58 events that gather statistics on the operation of the processor and memory system. Six counters in the PMU accumulate the events in real time. The PMU counters are accessible from either the processor itself or the external debugger [22]. Hardware measurements suffer from the obvious disadvantage of hardware availability requirement and assumes the running software is perfect and free of bugs.

3.5.2 Commercial solutions

Commercial solutions for cycle-accurate virtual prototyping of SoC platforms based on modern ARM processors such as Cortex A9 is provided by Carbon design systems [23], coMET from vaST systems [24] and processor designer platforms like coWare processor designers which allows you to describe the processor architecture in question using LISA [25]. ARM provides Realview ARMulator which is an instruction set simulator for ARM 7, ARM 9, ARM 10 and ARM 11 processor families [26]. Instruction set system models (ISSM) are provided for cortex-A8, cortex-M0,cortex-M1,cortex-M3 and cortex-R4 profiles. Real-time system models are provided with Realview development system (RVDS) professional edition and extends modelling support (not detailed implementation of hardware) to various emulation boards of several processors including Cortex-A9 [26]. Arm Profiler is a component of RVDS professional edition. ARM Profiler produces an analysis file containing detailed information on the executed code, such as call sequences for various functions, timing characteristics, cycle counts, and instruction counts[26]. It provides hardware profiling information subject to following restrictions:

- The ARM Profiler is not a debugger.
- The ARM Profiler does not track memory interactions. It knows when an opcode accesses memory and tallies the size of the memory access in the Accessed column of the table reports, but it cannot track whether the memory access is to a cache or a slow external memory.
- Only information on average cycles per instruction is provided.

3.6 Summary

This section summarises the various methods of execution time analysis listing its pros and cons.
High Level modelling languages:
SDF
+ Suitable for signal processing applications.
+ Suitable for critical cycle analysis, schedulability analysis (depending on availability of input tokens)
and rate-optimal buffer sizing.
- Fixed worst case execution time of actors (tasks) need to be given as input.
- Scheduling is determined by static order of actors as they appear in the SDF.

SADF
+ Addresses execution time variability due to dynamic inputs (scenarios). The execution times of actors need not be fixed.
+ Scheduling is influenced by dynamic inputs.
- Probability distribution of scenario occurrences and hence execution time distribution of actors need to be determined.

Basic block methodology:
+ Consists of suitable number of instructions to perform pipeline and cache analysis.
+ Branch prediction analysis has to be done just once per basic block as by definition, each basic block consists of one branch at the end.

Mathematical model
+ Generic, simple and least modelling effort.
- Inaccurate if program computation time is based on measured CPI (cycles per instruction) of one basic block when the instruction mix in the whole program is not similar to the instruction mix in the basic block.
- Inaccurate for modern processors which hide latencies due to cache misses very well. Thus, multiplying cache misses by respective miss penalties and adding it to the pure computation time leads to overestimation of the execution time.
- Inaccurate if branch mispredictions according to the branch prediction scheme of the

target processor is not incorporated in the model.

Hybrid methods - WCET
+ Increased accuracy of execution time due to pipeline analysis and cache analysis.
- Static pipeline analysis too complex for out-of-order processors.
- Dynamic data cache accesses using indirect addressing (via register contents) not considered. These accesses cannot be resolved for a hit or miss statically.
- Manual annotations of loop bounds have to be specified.

SystemC TLM
+ Separation of concerns such as functionality and timing.
+ Models expressed in systemC can be simulated and they are fast.
- Static computation of basic block execution cycle count.
- Dynamic data cache accesses using indirect addressing cannot be resolved statically for a hit or miss.
- Cache-level hierarchy not considered.

Instruction level functional simulation (Qemu, Valgrind, OVP Fast models)
- Timing of instructions is not modelled. Only the Instruction set architecture is modelled.
- Architectural features such as superscalar issue or out-of-order execution is not modelled for complex processors.

Hardware measurements and commercial solutions
+ 100% accuracy.
- Dependent on hardware availability.
- Porting problems for complex software.
- Expensive.

Algorithmic abstraction level using POOSL

4

4.1 POOSL vs SDF

SDF is a high-level modelling language for signal processing applications useful for analysis of task scheduling, buffer sizing and critical path analysis as explained in subsection 3.2.1. POOSL, as explained in subsection 3.2.2, is a specification language in the high-level modelling category which simulates a specification of application and hardware platform annotated with timing values. The merits and demerits of both the approaches were studied which is summarised below.

POOSL

1. More suitable for average performance metrics.
2. Application, Scheduler and hardware has to be explicitly modelled.
3. Rate monotonicity need not be preserved.
4. Analytical algorithms such as critical path determination, rate-optimal periodic scheduling cannot be applied in this simulation environment.
5. Provides a GUI environment and a flexible modelling framework. Can be easily adapted and parametrised.

SDF

1. More suitable for worst case performance metrics.
2. Scheduling is static and decided by the dataflow. Each actor has to be isolated and the hardware architectural influences have to be modelled around the actor.
3. Assumes applications are rate monotonic i.e if a token is delayed , output of the actor is delayed as well.
4. 'Hard' real time guarantees can be analysed and other analysis algorithms to find the maximum cycle mean and rate-optimal scheduling can be performed.
5. Any modifications without understanding the Dataflow model can lead to inaccurate results.

Due to reasons such as ease of use, flexible framework, GUI availability and shorter learning curve POOSL was chosen to be explored further. Both the modelling languages

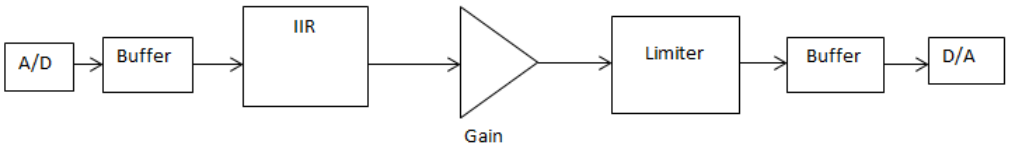


Figure 4.1: Algorithm block diagram

need accurate values for actor firing times or execution times of tasks. Below we describe a hybrid approach utilising performance counters on the hardware to measure execution times of algorithmic blocks and using POOSL to describe the application and hardware interface.

4.2 Hybrid approach using hardware and POOSL - algorithmic abstraction level

The chosen algorithm for this experiment was an IIR high pass filter, Gain and Limiter block as shown in figure 4.1. All the sub-blocks work on blocks of 32 audio samples. To simulate conditions of the periodic block sample interrupt triggered by DMA (refer subsection 2.1.1), a timer was configured and its period was set to the time period corresponding to the block sampling frequency (32* 1/sampling frequency). On expiry of the timer, a flag is set to boolean true. In the main loop, the flag is polled, and upon being set to true, it copies 32 audio samples to internal memory and starts executing the algorithm. The audio samples are placed in external DDR (Double data rate) memory.

4.2.1 Hardware/SDK setup

The hardware which we were working with was a zynq 7z020 also called the Zedboard. It consists of a dual core cortex-A9 and FPGA fabric. The algorithm was run on a single cortex-A9 core, bare-metal without any operating system. The Xilinx SDK ISE 14.3 was chosen for software development and creation of the ELF. Xilinx provided the arm-xilinx-none-eabi compiler tool chain, board support package and boot scripts for startup.

Xilinx recommends to start the design flow with their PlanAhead tools part of the ISE 14.3 if the design involves running a bitstream on the FPGA (also called Programmable Logic). The PlanAhead tools then allow you to add an embedded source to include the cortex-A9 in the design. To configure the hardware settings for the cortex-A9 which is also called the processing system, another tool called xilinx platform studio (XPS) is launched. Since our design does not deal with the FPGA, we launch the XPS directly. The Processing system configuration such as selection/deselection of I/O peripherals, memory configurations and clock speeds is done in XPS and the hardware design is exported to SDK. In SDK, a board support package project is created depending on the

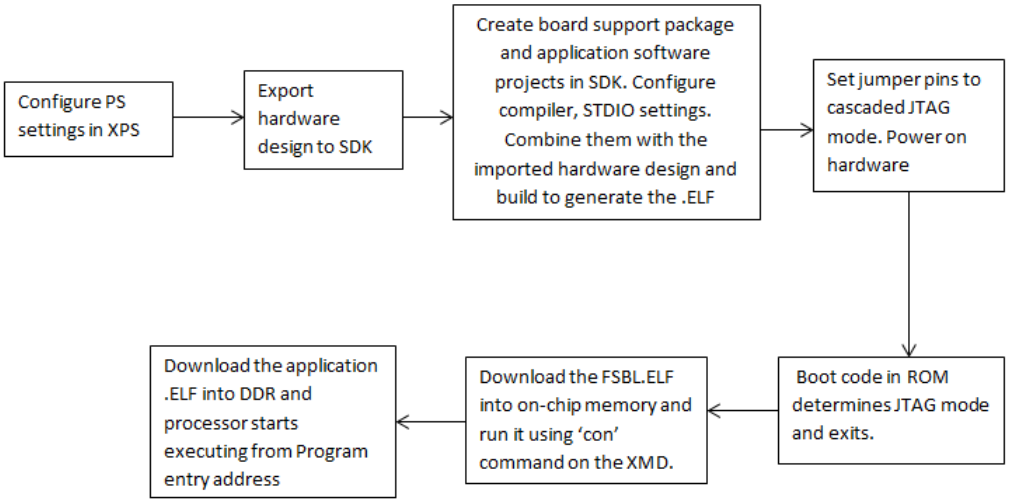


Figure 4.2: Zedboard design flow

desired hardware configuration: standalone (bare-metal), Linux (OS) or standalone_amp (Asymmetric multiprocessing mode). A software project with the code for our algorithm is then combined with the exported hardware design and board support package project and then built with the xilinx gcc tool chain.

On the hardware side, the following steps are performed to boot the cortex-A9 processor:

- Boot code in the ROM reads the mode pins and determines the boot mode which can be either JTAG, SD card, QSPI flash or from NOR memory.
- Once the mode is determined, the First Stage Boot loader is copied from the boot device or downloaded using the JTAG cable into the on-chip memory and executed. Xilinx provides the Xilinx micro debugger interface to download ELF files to hardware and issue other GNU debugger style commands.
- The First stage boot loader initialises the memory-mapped peripherals, clocks and the DDR. It then copies the first non-bit (bitstream) file in the boot device into the DDR. In case of JTAG, the application to be run on the processor is again downloaded using JTAG.
- Finally the program starts executing on the processor from the program start address optionally mentioned in the linker script/startup script.

The complete design flow is summarised in Figure 4.2.

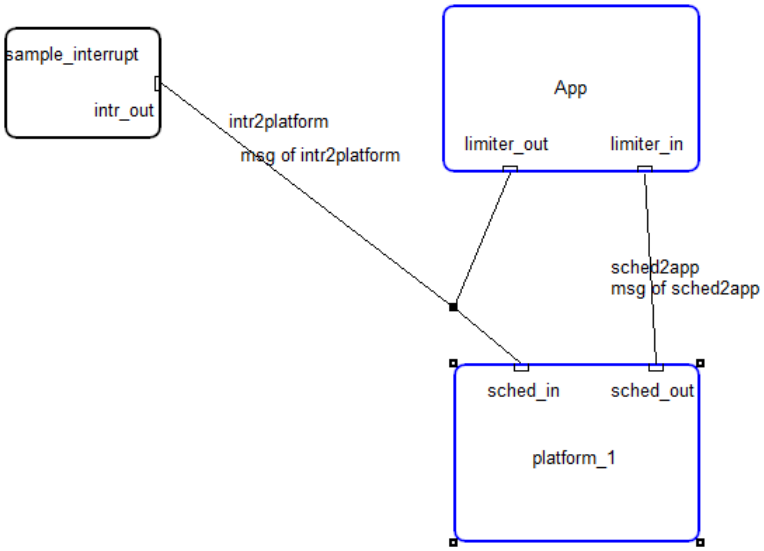


Figure 4.3: Top-level system design

4.2.2 POOSL setup

The SHEsim editor is used to develop models using POOSL. There are three hierarchical levels in POOSL model:

1. **Top-level** - This is the top-level view of the system. The interconnection of different clusters such as application, platform and any other external components of the system design is modelled in this level. Figure 4.3 shows such a top-level design for our experiment. The sample_interrupt is a process class which generates the timer interrupt once every 32 samples.The Application cluster consists of the algorithm blocks and the platform cluster consists of the scheduler and processor. The clusters are interconnected with channels and ports and they communicate across these channels through synchronous message-pairs which are mentioned above the channels connecting the ports in the Figure 4.3.

2. **Cluster level** - A cluster is an aggregation of processes and other clusters. Processes are the parallel entities in POOSL which execute concurrently. Processes can be connected to each other via channels and ports to exchange synchronous messages just like in clusters. Figure 4.4 shows the application cluster. The processes making up this cluster are memcopy, IIR filter, Gain and the Limiter blocks in the algorithm. They are connected to the cluster input and output ports named limiter_in and limiter_out in the figure. The algorithm blocks communicate only with the scheduler and do not communicate with each other in our design and hence there are no interconnecting channels between processes. Figure 4.5 cluster consists of the scheduler and processor processes. They communicate with each other via channels sched2core and core2sched.

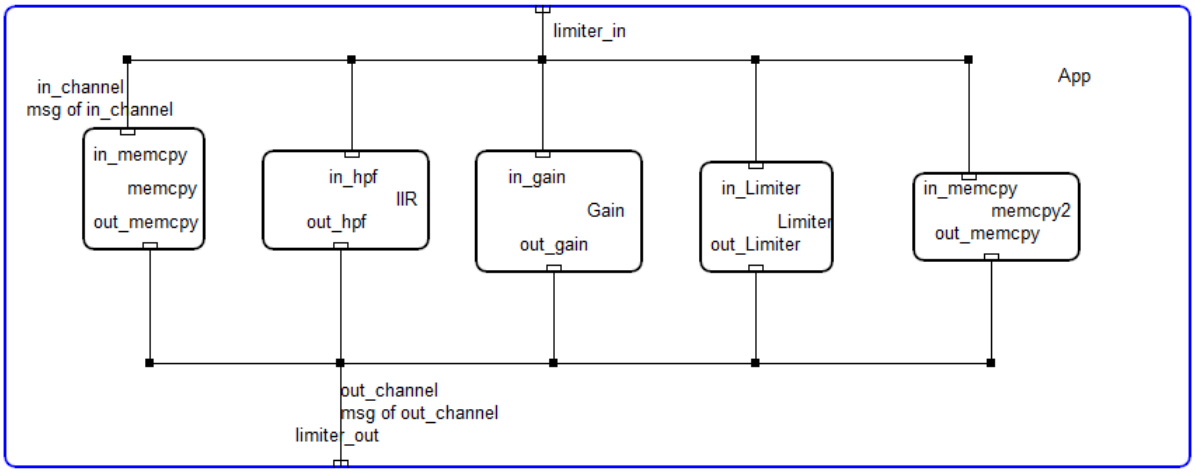


Figure 4.4: Application cluster

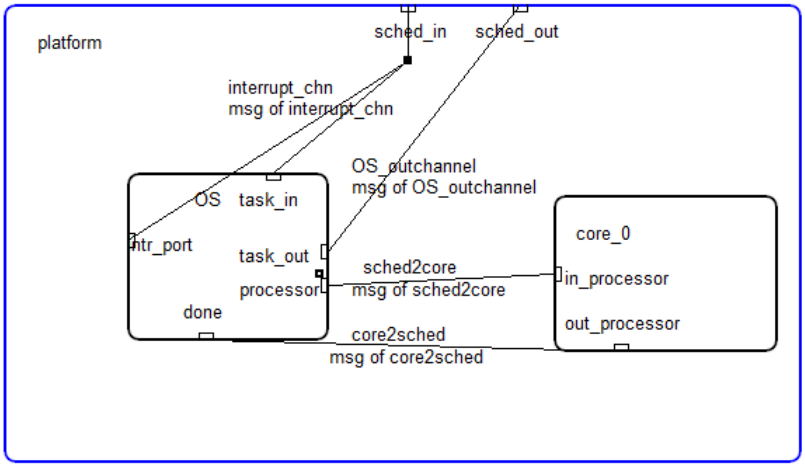


Figure 4.5: Processor cluster

3. **Processes and data object level** - Processes execute concurrently but each individual process is executed sequentially. A process consists of methods defining the process behaviour, instance variables of data objects, instantiation parameters for the process instance and an interface with other processes and clusters via channels and ports. Every process instantiates variables of certain data classes and is allowed to access only its own private data variables. Processes do not share data. While communicating with other processes, the sending process sends its private data as parameters of the message and the receiving process has to instantiate a variable of the same data object and receive the contents of the message in its own privately instantiated data. The message names correspond one-on-one at the sender and receiver process. To elaborate on synchronous message-passing: The sending process can resume activity once the message has been received by the receiving process. Similarly, the receiving process resumes activity once the

message has been received.

Figure 4.6 shows a snapshot of the process class description for processor. It consists of 2 methods processor() and initialize(). initialize() is the first method which is called when the process starts executing. The processor class has an instantiation parameter by name 'freq' which is the frequency of the processor which has to be set by the user. ID and ExecCyc are local variables of type integer which is a data class. in_processor and out_processor belong to the port interface of the processor for the incoming and outgoing messages of the processor.Incoming messages at the port are indicated by a '?' and outgoing messages are indicated by a '!'. Execute_task is an incoming message coming from the scheduler and done_processing is an outgoing message with no parameters to the scheduler. A delay() statement in POOSL is the only statement through which simulation time can be advanced. All other statements executed sequentially in a process or any synchronous communication which happens do not take up simulation time. In this example, the task execution time is indicated in the delay statement and execution of the statements following the delay() statement resumes only after the denoted time has passed. Also, the method processor() is called in a tail-recursive way in an infinite loop fashion.

Data objects/classes are passive sequential entities in POOSL. It consists of certain data (like attributes) and operations (methods) which can be performed on the data. Processes store this data by instantiating variables of those data classes. The instance variables can invoke data methods which get executed atomically and a result is returned.

4.2.3 The Approach

4.2.3.1 Hardware measurements

POOSL has been explored as a modelling and simulation tool for execution time analysis at the algorithmic abstraction level. The code structure for the chosen audio algorithm for the experiment has been modified to mimic realistic conditions of the DMA interrupt being triggered after copying a block of samples from the I2S buffers to memory. The execution times of the sub-blocks namely memcopy, IIR high pass filter, Gain and Limiter are measured by executing the block on hardware and instrumenting the code to read the cycle count register before and after the block starts and finishes execution. The measurements with the cycle count register showed a variation of 0.7% for 7000 values. A file containing 16,000 audio samples sampled at 16KHz was used as input. Each of the algorithmic sub-blocks were acting upon blocks of 32 audio samples and hence 500 runs of the algorithm were possible with 16,000 samples. Execution cycles were measured for each algorithmic sub-block for each of the 500 runs and the maximum value was chosen to simulate worst case behaviour. Caches have been disabled while measuring to simulate deterministic behaviour. However, branch prediction was enabled as the variability due to branch prediction across the algorithmic sub-blocks was much lesser compared to the case with caches enabled. Steps followed to perform hardware measurements are as below:

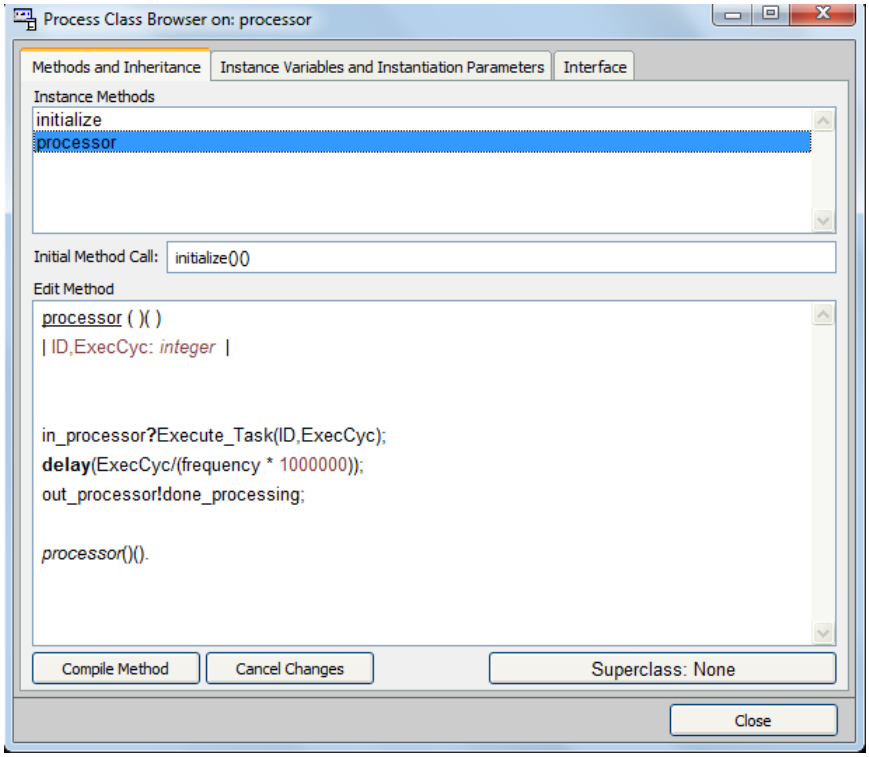


Figure 4.6: Process class of Processor

1. Change the processor mode to user mode to change performance counter settings.
2. Configure Performance monitor unit control register to reset cycle counter.
3. Clear interrupt enable and overflow status flags just to be sure we are not interrupted because of any overflow or performance counter interrupts.
4. Set the Count enable set flag to true.
5. Read cycle count register before and after the region of interest.
6. Clear the count enable flag.

Difference of the two captured cycle count values gives the measured execution time of the region of interest.

4.2.3.2 POOSL model

The POOSL model is adequately represented by Figure 4.7. It shows the communication between the processes in the POOSL model and also the pseudocode of the sequential behaviour of each process. A conscious decision was taken to separate responsibilities and encapsulate them into each process's behaviour. For example: the tasks have no knowledge about the task schedule. Only, the scheduler knows the task schedule and

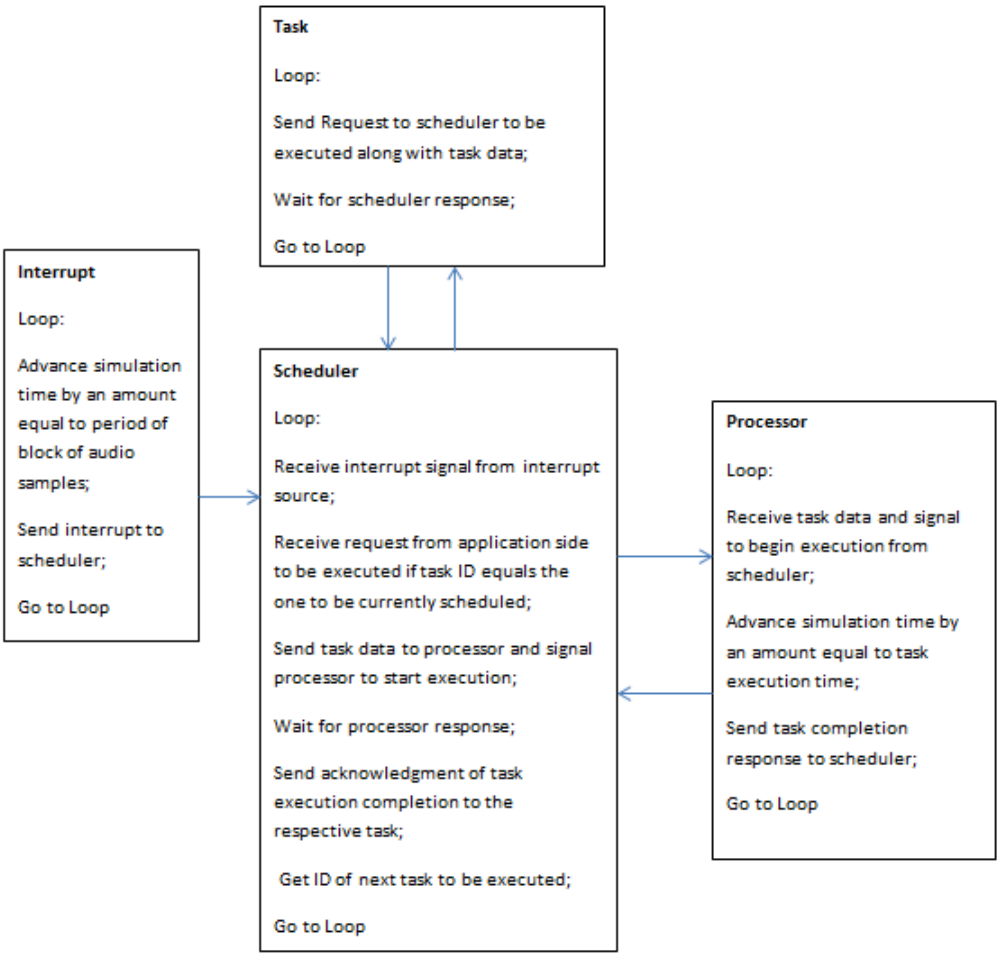


Figure 4.7: Overview of the communication in POOSL

receives requests from the tasks in that schedule order. Similarly, the processor is only responsible for executing the task at a certain frequency and signalling a response. This kind of separation of responsibilities makes the design more modular and easy to add or remove functionality specific to the component. Though the algorithm chosen is single-threaded and follows a static schedule of the IIR high pass filter followed by Gain and finally the Limiter, the scheduler was chosen to be added to the POOSL design to show how schedulers can be modelled for complex multithreaded audio algorithms. The POOSL model was simulated with SHEsim simulator and the results were evaluated against hardware for different program scenarios.

4.3 Results

The POOSL model was simulated and the execution time was analysed for two typical audio application scenarios:

No. of runs	Hardware	POOSL
1	107652	107903
10	1052245	1079030

Table 4.1: Results of Algorithmic abstraction level

1. One run of the whole algorithm
2. Ten runs of the whole algorithm. For example: for a multi-channel audio output, signal conditioning algorithms like the IIR high pass filter, Gain and Limiter might have to be executed for different blocks of audio samples for each channel.

Results are shown in table 4.1. The values correspond to execution cycle count for cpu running at 667 MHz. The observed difference in values between POOSL and hardware is due to pipelining effects. The assembly instructions used to read from the cycle count register are serializing instructions which means they are not pipelined. On encountering such instructions, all the prior instructions to the serializing instruction are executed and committed (written back to register files) before the serializing instruction enters the pipeline. Hence, when two consecutive blocks are executed without any serializing instructions, there's potential pipeline overlap which reduces the execution cycle count compared to the value got by adding the execution cycle count of individual blocks. Due to this effect, the POOSL model yields 2.5% overestimation compared to actual hardware. With this background, we are now in a position to critically evaluate this hybrid approach to execution time analysis at algorithmic abstraction level.

4.4 Evaluation

There are several advantages and disadvantages of choosing a high level of abstraction such as the Algorithmic abstraction level. Typical use cases of using such an abstraction level would be:

- To analyse deadline misses of a complicated multi-threaded audio algorithm with several smaller algorithm blocks and control flow dependent on incoming audio samples. The control flow can be easily represented in POOSL and file containing audio samples can be given as input. Sampling frequencies can be changed which result in different deadlines and the deadline misses can be checked for each scenario.
- To estimate how much processing can be done within the deadline. For example: How many IIR-Gain-Limiter chains can be executed before the next block sample interrupt occurs?

Using POOSL at the algorithm abstraction level requires very little modelling effort. It provides a quick way to evaluate if the complicated algorithm with several blocks and possibly an RTOS scheduler will adhere to deadlines or if buffer sizes are optimal.

POOSL can be very beneficial if an audio library is maintained and all the code blocks for all algorithms are chosen from this library. The code blocks of the library have to be measured and modelled only once and they can be reused in a variety of other audio algorithms. Also, the rich GUI of the simulator gives a visual interface which allows easy understanding of the algorithm flow. The simulation can be stepped through either with time steps (specified by delays) or communication steps (specified by synchronous message-pairs). A scheduler with all the required functionality for analysis can be easily modelled hence facilitating schedulability analysis of the algorithm if it is multi-threaded. The most important contribution of POOSL is its ability to specify parallelism. Using the 'par' primitive in POOSL one can specify the methods that need to be called in parallel. Thus POOSL is an effective vehicle to exploit and test parallelism in audio algorithms.

The biggest disadvantage of performing execution time analysis at the algorithm abstraction level is its rigidity. Measurements have to be repeated if the block size is changed. Also, dynamic features of modern pipelines like caches and deep pipelining introduce inaccuracies in results if they are not accounted for in the analysis. This calls for cache models and pipeline analysis to be integrated into the POOSL model which is difficult as it requires visibility of the instructions and data accesses. If the code structure of the algorithm has a lot of branching, branch prediction can also introduce inaccuracies into the POOSL predictions.

5 Basic block abstraction level

5.1 Introduction

The previous chapter exposed certain disadvantages of performing timing analysis at a higher abstraction level such as the algorithm sub-blocks. Hence, a lower abstraction level is required to improve flexibility and accuracy. This chapter introduces timing analysis at basic block abstraction level. A basic block as defined in section 3.3 is a block of instructions with one entry and one exit point for the control flow. The block of instructions can belong to the source code or corresponding assembly instructions.

Performing analysis at the basic block level eliminates the disadvantage of repeated measurements for a changed block sample size. Since the measurements are now done at the basic block level, the frequency of execution of the basic block can be changed and the execution cycle count will scale accordingly. For instance, the IIR high pass filter consists of loops which were run for 32 iterations corresponding to 32 samples. The measurements of the sub-blocks taken on hardware were for 32 sample block scenario and has to be taken again if the block size is changed to 64. Whereas, if you are working at the basic block level, the loop body which is the basic block has to be measured and it can be multiplied by the desired block sample size which is the iteration frequency. Another big advantage of working at basic block level is a more accurate pipeline overlap and cache analysis can be performed at the basic block boundaries as there is more visibility in the code structure, number and type of instructions in each basic block. Branch prediction can be done once per basic block because by definition there exists a branch only at the exit point of each basic block.

Since we were dealing with hard real-time systems, worst case execution time analysis becomes crucial. To explore the different possibilities for timing analysis at the basic block abstraction level an experiment was devised. A worst case execution time analysis was compared to results of employing POOSL for execution time analysis, both analyses done at the basic block level. The accuracy of the time estimates provided by the two methods were compared against that derived from a cycle-accurate simulator. A commercial WCET tool called aiT was chosen which is developed and marketed by absInt Gmbh in Germany. Their target customers are aircraft and automotive domains. It is available for free evaluation with an academic license. The ARM platform chosen was cortex-M3 and cycle-accurate simulator was Keil, by Arm Ltd. The chosen code for the experiment was a fixed point forward fft function which is one of the most important blocks of any audio algorithm. The code can be found in the appendix.

why aiT? A lot of WCET tools which were either open source or could be obtained with an academic license were investigated for this experiment. The considered options

were

1. SWEET(SWEdish Execution time analysis tool) developed by Malardalen university in Sweden.
2. Chronos developed by NUS, Singapore.
3. Heptane.
4. Bound-T.
5. RapiTime.

SWEET uses a low-level hardware analysis tool as a last step in the WCET estimation which is not available for download hence making it unsuitable for our analysis. Chronos supports only MIPS binaries for analysis and we were only interested in ARM cores. Bound-T has support only for ARM7TDMI cores and works very similar to aiT which supports cortex-M3 and cortex-R4F processors which are more advanced. There was no support or response from the Heptane and RapiTime groups. Thus, aiT, which also comes with a webEx tutorial and has an interactive GUI, was finally chosen for the experiment.

Why Cortex-M3? aiT provides analysis only for cortex M3 OR cortex-R4F processors with one academic license. The chosen cycle-accurate simulator, Keil supports both processors and hence a decision had to be made. Cortex-R4 processors have more number of pipeline stages than cortex M3 and consists of branch prediction unlike cortex-M3.

Cortex-M3 is a 32-bit microcontroller belonging to ARMv7-M architecture. It consists of a 3 stage pipeline as shown in figure 5.1. It is important to note that Execute and Writeback of results completes in the same cpu cycle. Hence stalls due to data dependencies for the incoming instructions into the execute stage in the next cycle are avoided. This significantly reduces efforts required for pipeline analysis for the cortex-M3. Execute stage consists of Multiply and Divide, Shift, ALU and branch, Address decode phase and load/store phase. It is a single-issue pipeline and hence only one instruction can occupy the execution stage at any given point of time. Hence structural hazard stalls for say instruction (i+1), ready to be dispatched into the execution stage, overlap with the execution cycles of previous instruction (i) observed with Keil. Cortex-M3 has no caches, MMU or floating point unit. Branch prediction is not present so indirect branches take 3 cycles to execute. 1 cycle is to determine the branch address and 2 cycles are for fetching and decoding the instruction at this address. Absence of caches and branch prediction simplifies analysing cortex-M3 further.

Thus, pipeline analysis is implicitly handled by Keil thanks to the simple 3 stage pipeline of cortex-M3. This becomes important as Keil does not monitor the pipeline. Choosing cortex-R4 where the execution and writeback happen in separate stages would require explicit pipeline analysis to detect stalls due to data dependencies. Since we did not have hardware available to perform measurements, there was no accurate source to validate the pipeline analysis. The focus of the experiment was to test the methodology

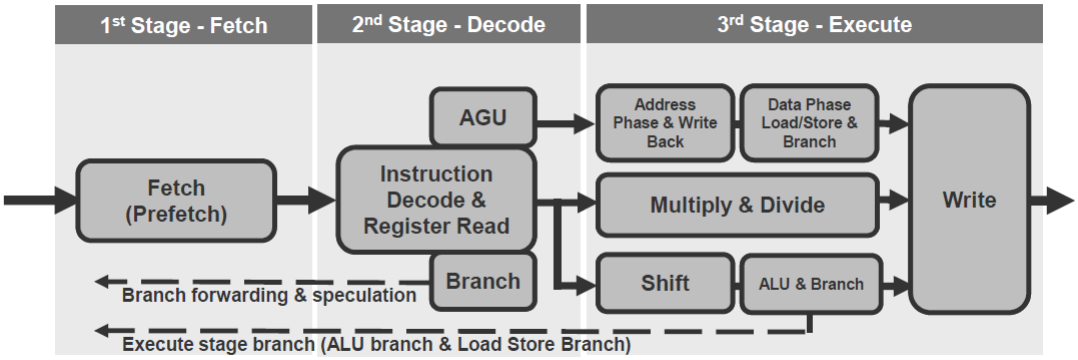


Figure 5.1: Cortex-M3 pipeline. Source: [6]

and hence the easier processor to analyse which in our case was the cortex-M3 was chosen. Also, prior experience with cortex-M3 influenced this decision.

5.2 Cycle accurate simulator - Keil

Keil is a cycle-accurate instruction set simulator from Arm Ltd. It can simulate cortex-M, cortex-R4, ARM7 and ARM9 devices but does not provide support for cortex-A processors. Keil also provides an IDE integrated with an ARM compiler to aid software development for the supported microcontrollers. Hence, its two main functionalities are as a debugger during software development or as a simulator depending on hardware availability. We used Keil in the simulator mode for this experiment to simulate cortex-M3 as hardware was unavailable.

Some of the first few steps in using keil is to choose a target microcontroller upon creating a new project which links the corresponding startup files with your application project. Other options such as frequency, RAM and ROM sizes required by your application should be configured as well. Our code and read-only data reside in the ROM space available on chip (0x0 to 0x20000000). No external memory was used. The simulator mode is enabled in the debug options and a debug session is started. Breakpoints are placed at the assembly instructions corresponding to entry and exit of basic blocks in the disassembly to measure the execution cycles. The register field STATES under category 'internal' in the Registers window shows the execution cycles of the cpu in simulator mode. The entire code was broken into basic blocks and the STATES register field was used to measure execution cycles of each basic block which were required as input for the POOSL program modelling.

5.3 Algorithm modelling at basic block level in POOSL

5.3.1 Approach

POOSL offers the perfect platform to model an audio algorithm at basic block level. It can also be done in systemC but we chose to do it in POOSL since we were already acquainted with it. POOSL is very expressive and offers all the required constructs to express the control flow of audio algorithms such as loops, function calls, decision statements, abort and interrupt primitives. The execution times of basic blocks can be mentioned with delay statements. For our experiment, we created a process for the fixed point fft function. An overview of the division of basic blocks for a part of the fft function is shown in figure 5.2 and the associated control flow graph in figure 5.3. The control flow of the basic blocks were specified using available constructs and primitives in POOSL. Since we were only interested in the execution time analysis and not the functionality of the algorithm, the functionality of the basic blocks were not specified. Hence, the basic blocks were specified only with delay statements containing execution times which were measured in Keil. Pipeline overlap was not noticed across basic blocks when they were timed separately and together using Keil and hence were ignored in our POOSL model. Cache and branch prediction models were not necessary as the cortex-M3 does not possess these dynamic features. Important parameters like fft size and frequency of processor were mentioned as instantiation parameters for more flexibility.

Figure 5.2 shows a part of the fft function expressed at basic block level in POOSL. As mentioned before, the control flow is expressed separately with the while and if constructs. Notice the statements altering values of local variables on which iteration frequencies, increments of loop counter variables and conditional constructs such as while depend on. The expressive nature of POOSL even allows us to perform bitwise operations on integer variables which is crucial as the values of these variables alter control flow. The advantage of this expressiveness is more apparent when we discuss static WCET analysis in the next section.

5.3.2 Advantages

The advantage of the basic block abstraction is two fold. Analysis has to be performed at the basic block level instead of instruction level and also simulation speed is increased if you have fewer delay statements, one per each block instead of each instruction. Furthermore, advantage of using POOSL for basic block modelling is the ability to separate functionality and timing. The POOSL statements used to express the control flow get executed but do not add to simulation time and hence do not affect our basic block execution measurements. Timing can be explicitly mentioned with delay statements along with the functionality of the block if desired. Also, cache, branch prediction and pipeline analysis models can be easily integrated in POOSL and has to be done just once per block. Algorithms specified at the basic block level can be combined to form more complex algorithms with higher flexibility and accuracy for execution time analysis.

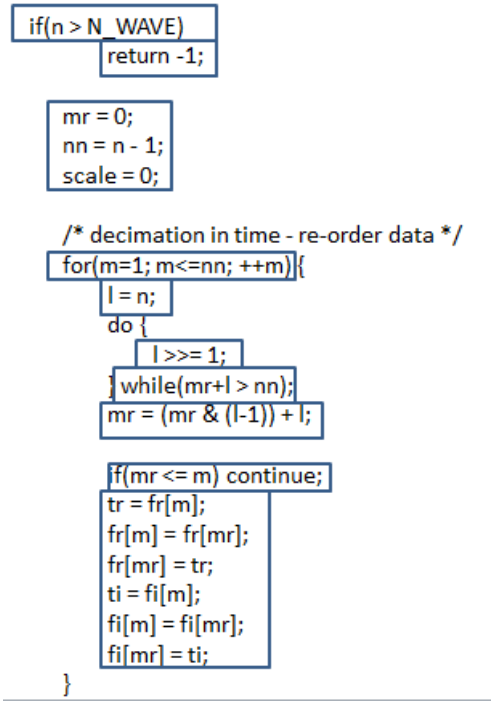


Figure 5.2: A part of the fft function divided into basic blocks

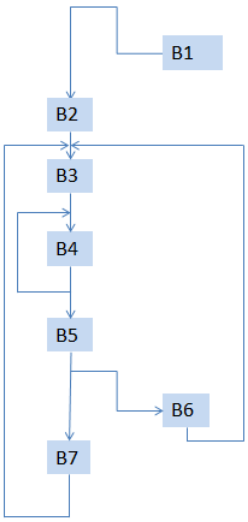


Figure 5.3: Control flow graph of basic blocks of the part in 5.2

5.4 WCET analysis using aiT

5.4.1 Static Analysis

WCET using the tool aiT uses static analysis of the source code of a program to predict the Worst case execution time. Static Analysis of the source code refers to determining the WCET of the code without executing it on the hardware, instead predicting the worst case behaviour of the program by analysing the source code and using an accurate hardware model which models all possible hardware effects that can lead to worst case scenario. The hardware model offers the flexibility to analyse scenarios which are not possible or may not occur while using measurement techniques. Static analysis is also especially useful when the execution paths in the program varies according to the inputs to the program. This execution time variability is avoided by using static analysis because static analysis follows certain rules to determine the WCET of the program for *any* given input. The rules of static analysis for resolving if else constructs are as follows:
Max(T(if) + T(test), T(else) + T(test)) as the resulting WCET for an if-then-else block where T(if) and T(else) is the execution time of the body of the if and else constructs respectively. T(test) is the execution time corresponding to the condition check of the if-then-else block.
For an if-then block, the worst case from an execution time point of view is assumed, which is the 'if' condition being true and T(if) + T(test) determines the WCET of the if-then block. With this introduction of static analysis we can now shift our focus to aiT.

5.4.2 aiT working phases

Source: AbsInt Angewandte Informatik GmbH, Safety Manual for aiT, Revision 217166 of April 2, 2014, used with permission. Figure 5.4 shows the aiT workflow stages. The workflow consists of 4 phases.

The first phase is called the decoding phase where the control flow graph of the program is determined by reading the binary executable and expressed in an intermediate format called CRL which stands for control flow representation language. The CRL format aids further transformation such as loop transformation. The control flow graph is then reconstructed with all the transformations and passed to the next phase.

The next phase performs microarchitecture analyses such as combined loop-bound and value analysis and cache/pipeline analysis. Value analysis tries to determine approximate values in registers, memory cells for each program point and execution context by using *abstract interpretation* which is mathematical semantics to be followed for abstract execution of instructions in static program analysis. This approximate information helps in finding loop iterations and memory addresses of indirect memory accesses which can aid cache analysis. The loop-bound analysis tries to determine upper bounds on loops from the CRL of the previous phase and also from the output of value analysis. If the loop-bounds cannot be determined by aiT, user annotations supplied by the user will be used to determine these bounds. Cache/pipeline analysis is performed by a single analyzer. Instruction sequences are fed from the control-flow graph to a timing model

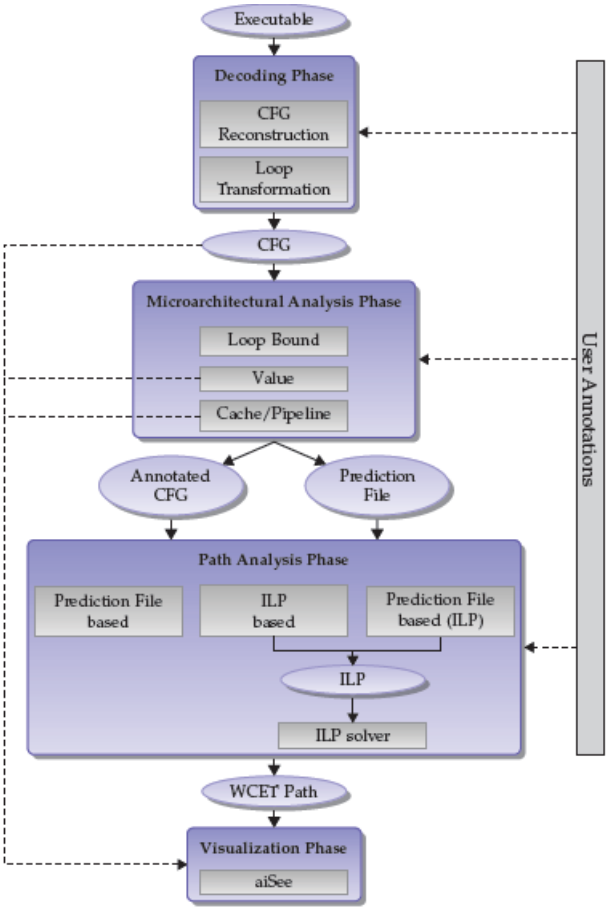


Figure 5.4: aiT workflow. Source: AbsInt Angewandte Informatik GmbH, Safety Manual for aiT, Revision 217166 of April 2, 2014, used with permission

and the cache/pipeline analyzer tracks system state changes due to pipeline and cache changes and the elapsed core clock cycles are recorded. The output of this phase which is used by the subsequent path analysis phase is a prediction graph containing details of the loop-bounds, value analysis and system state changes due to cache/pipeline analysis.

The path analysis phase is used to combine the results of all the previous phases to determine the WCET of the worst-case path which is annotated with execution times of the basic blocks. The WCET is determined by formulating an ILP (Integer Linear Program) with the execution time being the maximizing function and the constraints of the ILP are given by the control-flow graph of the worst-case path. ILP solver is used to solve the ILP and display the WCET to the user.

The final phase is the visualization phase where the results are displayed to the user along with a graphical representation of the control-flow of the worst-case path as seen in figure 5.5 for fft size 32. The complete analysis is also presented in a human-readable format in a text file.

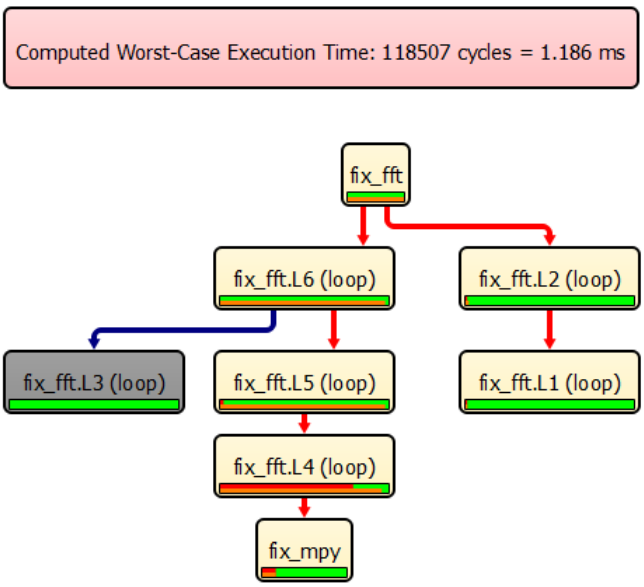


Figure 5.5: WCET using aiT for fft size 32

5.4.3 Manual annotations

aiT allows users to provide information which cannot be determined automatically with aiT analyzers through annotations. The annotations must be specified according to a defined standard called AIS, developed by AbsInt. Upper bounds on loop iterations, indirect function calls through function pointers and memory access times to external memory can be mentioned through these annotation files. We mentioned the compiler used, frequency of core, memory ranges for read-only and read-write memories in the annotation file.

For our experiment, we started with unrolling the loops by a huge value of 100. This unrolling factor can be mentioned in the annotation file. This resulted in the analysis not ending even after 1.5 hrs. This was the case for any value between 64 and 100. When we lowered the unrolling factor to any value lesser than 64, the analysis ended in errors asking the user to specify loop bounds. The error occurred because value and loop-bound analysis failed to determine the loop bounds. The number of loop iterations of nested loops in the fft function are not fixed and change based on intermediate values of certain variables in each iteration of the outermost loop. Hence, it becomes difficult to mention one fixed upper bound for the loops. We tried to force value analysis to evaluate contents of certain registers holding the intermediate values of the variables to determine the loop bounds but it resulted in errors. The only solution was to provide with the upper bounds for all loops in the user annotation file and determine the level of overestimation in the WCET. The various loops in the pseudocode 5.4.3 of the fft function are named in the

comments as they appear in figure 5.5.

```
nn = n-1;
for(m=0;m<=nn ; m++)//loop 2
{
  //body of loop
  do{
    //body of do-while
  }while(condition)//loop 1
}
.
.
.
.

l=1;
While(l<n)//loop 6
{
  if(inverse)
  {
    for(i=0;i<n; ++i)//loop 3
    {
      //body of loop
    }
  }
  else
  {
    //body of else
  }

  istep = l << 1;
  for(m=0;m<l;++m)//loop 5
  {
  //body of loop
    for(i=m; i<n; i+=istep)//loop 4
    {
      //inner loop body
    }
  }
  l=istep;
}//end of while
```

'n' in the above code stands for fft size. In actual execution, loop 3 is never reached as we are interested only in forward fft and thus the 'inverse' variable is made zero. We force this path to be termed infeasible marked as grey in figure 5.5 by altering the loop bound in the annotation file.

Consider the case when 'n' is 32. The influence of variables 'istep' and 'l' on the control flow and also on the loop iteration count can be clearly seen in the code. Value and loop-bound analysis failed to analyse registers for these changing values and consider loop iterations for different loop contexts. Hence, maximum loop bounds for loop 6 was mentioned as 5, loop 5 as 16 and loop 4 as 15 in the user annotation file. In reality, this is the loop iteration count for just one execution context when l=16 for loop 5, and

FFT size	aiT	Keil	Basic Block in POOSL
16	25603	5365	5345
32	118507	12765	12782
64	591696	29309	29633

Table 5.1: Results of Basic Block

when m=0 and istep=2 for loop 4. Hence the actual worst case loop iterations of loop 4 in each iteration of outermost loop 6 and when m=0 is 15+8+4+2+1 = 30, but aiT overestimates this value to be 975 because it considers the maximum loop iterations of all outer loops and inner loops specified in the annotation file for every execution context of loop 4. This was observed to be the biggest drawback of the WCET estimation using aiT.

5.5 Results and Observations

Table 5.1 presents the results of the basic block methodology experiment. The values are the cpu execution cycle count when the clock frequency was 100 Mhz and the compiler used was the arm compiler. aiT shows a 20x overestimation whereas using POOSL resulted in a worst case of 1.1% error for size 64. Static analysis clearly has a big disadvantage if the code structure employs complicated nested loops whose loop iterations change according to values of variables, hence making it unsuitable for audio algorithms where such complexities exist.

On the other hand, POOSL reports much better results for a fairly simple microcontroller like cortex-M3. Working at the basic block level in POOSL makes it easy to parametrise the audio algorithm and change loop bounds, sample frequencies and input data conveniently. On the downside, to use basic block modelling with POOSL for a complex processor like cortex-A9 will require an accurate cache, pipeline and branch model integrated into the basic block model of the source program for more precise execution time analysis. Also, modelling at the basic block level requires significant modelling effort if the source program is huge. Hence, it becomes a viable option only when the code blocks thus modelled are reused for a large number of audio algorithms.

Cycle-accurate abstraction level

6

6.1 Introduction

The previous chapter explored basic block methodology for performing execution time analysis. Modelling at the basic block level in POOSL for cortex-M3 provided accurate results compared to Keil. To use the basic block methodology for modern processors such as cortex-A9, one also needs an accurate model for cache, branch prediction and pipeline analysis to combine with static execution times of basic blocks. Modern processors pose different challenges to employ basic block methodology for execution time analysis. Indirect addressing using addresses in registers makes data cache accesses hard to analyse. Detailed knowledge of the microarchitecture is required to perform adequate pipeline overlap analysis. Data dependences between instructions need to be known to detect stalls. Advanced branch prediction schemes allows speculative execution and knowledge of these schemes is required to determine the number of branch misses and pipeline flushes. Thus, these challenges can be tackled only at a lower abstraction level - the cycle-accurate instruction level.

There is a lot of ongoing research to address the complexities posed by modern processors in the field of execution time analysis and WCET determination. More detailed systemC models for cache, branch prediction, pipeline, underlying memory system have been developed like in[27]. Complexities of modern processors such as speculative execution, execution context and pipeline overlap have been addressed in [28]. Current research on WCET analysis is employing detailed pipeline models and flow analyzers to alleviate obvious problems of static analysis which aggravate when used with modern processors [10]. We employ a very straightforward approach to solve the problems associated with modern processors. The solution is to use a computer architecture simulator.

A computer architecture simulator is used in computer engineering research to experiment with processor architectures. A typical use case would be to develop better designs for improving performance of the processor. It simulates the behaviour of the underlying architecture of the processor including pipeline stages, caches and branch prediction. For our experiment, we chose the Gem5 simulator[29], a computer architecture simulator resulting from a contribution of several universities and companies. Gem5 provides support for Armv7-A ISA and hence can be used to simulate a cortex-A9 core, making it the perfect choice for our thesis. The other computer architecture simulators which were considered were SimpleScalar, ESESC, Multi2Sim and OVP processor models. SimpleScalar, developed by Todd Austin while he was a Phd student at University of Wisconsin, models a strongARM core and does not support Armv7-A ISA. OVP fast processor models by Imperas ltd. models cortex-A series processors but microarchitecture of the processor

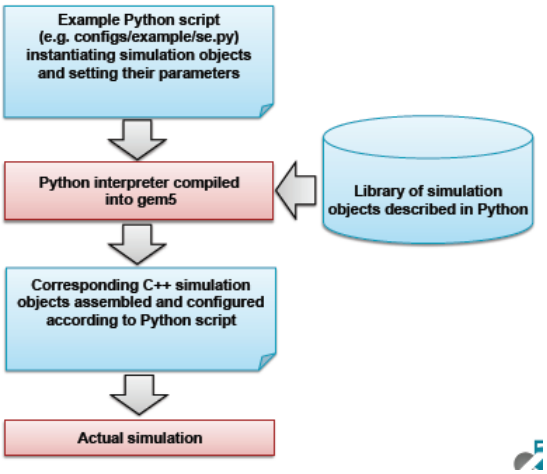


Figure 6.1: Workflow of gem5. Source: [7]

is not modelled. Also, OVP models are for functional simulation of instructions, hence they are not cycle-accurate which makes it unsuitable for our analysis. ESESC stands for Enhanced SESC which is an extension to the SESC simulator developed at University of California, Santa Cruz. It does not support full-system simulation (previously there was support for this) and it sacrifices a small amount of accuracy to improve simulation speed as it employs time-based sampling. Time-based sampling is where the timing within a sample is modelled cycle-accurately, whereas between intervals functional emulation is employed [30]. Hence ESESC is not suitable as our target was a simulator which is completely cycle-accurate. Multi2sim is a framework to support cycle-accurate simulation of heterogenous multiprocessors. It currently provides only functional emulation model for ARM cores but not the cycle-accurate architecture model hence deeming it unsuitable for our analysis. Thus, Gem5, which not only models the architecture but also the peripherals and underlying memory system in a SoC (full-system simulation), was chosen. Another reason to choose this simulator was an active mailing list and user community.

6.2 Background and working

Gem5 is a result of merging two simulators M5 and GEMS [31]. M5 was focused on full-system simulation comprising of pipelined cpu-models, OS support and I/O support. GEMS was focused on multi-level memory-system models which aided research on memory hierarchy. Thus, combining the two simulators makes Gem5 powerful enough to simulate a complete SoC making it the most desirable option for design space and architectural exploration. Gem5 software is written using C++ and python. Every component which is simulated in Gem5 is an object. The C++ classes define the behaviour of different simulation objects and Python scripts are used to configure, instantiate simulation objects, connect different simulation objects to each other and start the simulation. Every object inherits the SimObject base class. Also, Base classes are defined for all the major components of the system (CPUs,Buses,Caches etc..) and more specialised com-

ponents inherit from this base class (out-of-order cpus, i-cache, interconnects etc)[31]. There are python scripts for user-level configuration of these major components according to the system they want to simulate. 6.1 shows the basic workflow of Gem5. Gem5 is an event-driven simulator. All objects schedule their own events. Time is measured in 'ticks'. Each 'tick' is configured to be one picosecond. Based on your core frequencies, every clock period corresponds to certain 'ticks'. Based on their clock domain, objects use corresponding 'ticks' to schedule events.

6.2.1 Gem5 options

Gem5 is a higly flexible yet detailed simulator. It provides several options to the user to configure their target system as close to the real system as possible. Below are certain configurable options:

1. **Binary** - Depending on the speed of simulation, debug and tracing facilities desired the Gem5 binary can be built as:
   - gem5.debug - Debug build consisting of tracing and assertions.
   - gem5.opt - Optimized build consisting of tracing and assertions.
   - gem5.fast - Optimized build with no debug or tracing support.
   - gem5.prof - Same as gem5.fast binary but includes profiling support.
2. **Simulation mode** - Gem5 offers two modes for simulation. Full-system simulation which is capable of booting operating systems also models the bare hardware, peripherals, interrupts, priveleged instructions and exception handlers. Gem5 models two SoCs containing ARM cores, the Realview and Versatile boards containing cortex-A9 and cortex-A15 cores respectively. By default, the realview PBX machine type is chosen if full system simulation is preferred. For our Gem5 experiment, we use full-system mode with the machine type as Realview in a bare-metal environment. Syscall emulation (SE) mode models ISA and common system calls, which are emulated by calling host OS system calls. SE Does not model detailed virtual memory management and uses simpified address translation.
3. **Cpu type** - Gem5 offers a wide variety of cpu models and cpu types based on the desired level of accuracy. The available cpu types are 'AtomicSimpleCPU' which performs all cpu and memory operations for an instruction in one cpu cycle. 'TimingSimpleCPU' introduces some delays in the memory operations. 'InOrder' and 'O3' (Out-of-Order) cpu types which are more detailed and cycle-accurate. We use the 'arm-detailed' cpu type which is an O3 cpu model modelling the ARM core in specific.
4. **Memory type** - Gem5 consists of 2 memory configurations Classic and Ruby. Classic is slightly less detailed and accurate than Ruby. Ruby is more useful if complex cache coherence protocols need to be configured. We use the Classic memory as

Ruby full-system is not supported with arm-detailed cores. Classic memory comprises of a simpleDRAM memory controller and supports DDR3, LPDDR2/3 external memories. The memory type we use in this experiment is an lpddr2_s4_1066_x32 which comes closest to the DDR3_1066_x32 on the xilinx zedboard which was our target system. Gem5 also allows you to run simulations without caches, with only L1 cache or with multi-level caches.

Thus the following options were specified on the command line for simulating a full system similar to the zedboard.

```
build/ARM/gem5.debug Configs/example/fs.py --cpu-type=arm_detailed -n 1
--sys-clock=667MHz --cpu-clock=667MHz --mem-size=256MB
--mem-type=lpddr2_s4_1066_x32 --caches --l2cache --l1d_size=32kB
--l1i_size=32kB --l2_size=512kB --l1d_assoc=4 --l1i_assoc=4 --l2_assoc=8
  --cacheline_size=32 --kernel=name-of-elf.elf --bare-metal --dtb-filename=none
```

-n 1 indicates a single core, –sys-clock is the system clock for system buses, on-chip and off-chip caches. On the zedboard, since we do not use any memory-mapped peripherals for this experiment which run at the peripheral clock. Everything that was of interest to us such as the core and buses to on-chip and L2 caches run at the same core clock frequency. Thus, the sys clock and cpu clock are set to 667MHz which was the core clock frequency on the hardware. The sizes and associativity of caches can be mentioned on the command line as well.

6.2.2 Important features

Gem5 has several interesting features which aid in execution time analysis and performance modelling. It allows the user to place checkpoints in the source code around the region of interest and restart simulation from the checkpoint at a later point in time. On the debugging front, Gem5 has put in a lot of useful print statements which can be printed onto the console or copied to a trace file by using debug flags on the command line. Tracing can be enabled to trace the execution of instructions through the pipeline stages. The most important feature is the collection of various statistics of interest during the simulation which is output to a text file in the output directory. The statistics cover a wide range from number of individual type of instructions executed to cache misses and branch mispredictions. One can also add their own statistics to Gem5.

6.3 Configuration

The most important and difficult part of execution time analysis using Gem5 is configuration. This is because Gem5 offers cycle-accurate behaviour of all the modelled components but the timing parameters involved in the simulation need to be configured by the user according to the target system. Our target system is the xilinx zedboard consisting of a dual core cortex-A9. We focus on a single-core and hence cache-coherence, bus and DRAM controller arbitrations are ignored. Since, the target hardware was available to

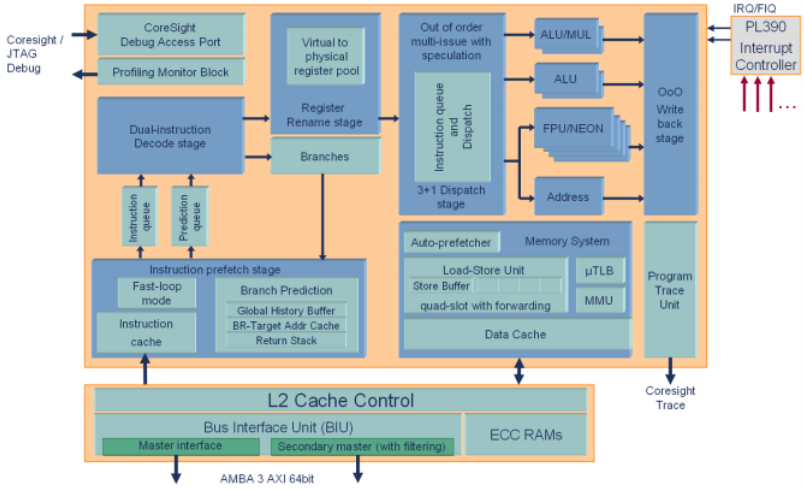


Figure 6.2: Cortex-A9 pipeline. Source: [8]

us we were able to compare the final results on Gem5 and hardware and thus validate our configuration.
We started the configuration process by making sure we were running the same binaries with identical hardware settings on both Gem5 and hardware. We used the SDK provided in the xilinx ISE 14.3 to generate the ELF for the source code. The tool chain used was arm-xilinx-none-eabi. The same boot scripts, linker scripts including stack and heap initializations, compiler flags and mmu page tables were used for both Gem5 and hardware binaries. The elf's were compiled with the static flag to generate statically linked executables.
Certain hardware settings were configured on both Gem5 and the hardware to make the analysis easy to compare such as Exclusive caches (a copy of data is either in L1 or L2 but never both), both L1 and L2 cache policies were set to write back write allocate, SMP (Symmemtric multiprocessing) flag was disabled, prefetching of cache lines was disabled and flow prediction (branch prediction) was enabled.

6.3.1 pipeline, cache and TLB configuration

Figure 6.2 shows the cortex A9 pipeline. The following changes shown in table 6.1 were made to python script O3_ARM_v7a.py in configs/common directory.

Some parameters were commented out such as write buffers and size for TLB cache. Also, the TLB cache associativity mentioned was for second stage TLB and hence the is-top-level parameter was made false. The sizes for the microTLB (first stage) and second stage TLB were set as 32 and 128Kb respectively in src/arch/ArmTLB.py and the gem5.debug executable was recompiled. The rest of the configurations were unmodified.

Configuration parameter	Gem5 default	Modified for cortex-A9
LQEntries	16	1
SQEntries	16	4
fetchWidth	3	2
fetchBuffer Size	16	8
decodeWidth	3	2
renameWidth	3	2
issueWidth	8	2
dispatchWidth	8	4
commitWidth	8	4
squashWidth	8	4
wbWidth	8	4
numPhysIntRegs	128	56
numIQEntries	32	16
I-cache tgts-per-mshr	8	2
D-cache mshrs	6	4
D-cache tgts-per-mshr	8	4
D-cache write buffers	16	4
TLB cache assoc	8	2
TLB cache is-top-level	'true'	'false'
L2 cache mshrs	8	4
L2 cache write buffers	8	3

Table 6.1: Configuration changes for pipeline, cache and TLB

6.3.2 Functional units, operation latencies and branch predictor

The count parameter of functional units in O3_ARM_v7a.py was changed from 2 to 1 for Integer ALU and floating point functional units after observing a configuration done for samsung exynos (refer Appendix section 8.2) found at [32]. Also, the separate load store functional units were merged into one as this is the case in cortex-A9. Some of the operation latencies were changed in accordance with the technical reference manuals of cortex-A9. In the branch predictor functional unit settings in O3_ARM_v7a.py, choicePredictorSize and globalPredictorSize were reduced to 4096 from 8192 as the global predictor size in cortex-A9 is 4096 entries. The BTB cache entries and RAS size were reduced from 2048 to 512 and 16 to 8 respectively.

6.4 Fine tuning

Some more fine tuning of configuration was done with respect to bus widths, external memory parameters and cache ports. In file DRAMctrl.py under src/mem the following changes 6.2 were made to the class LPDDR2_s4_1066_x32. On the zedboard there are two DDR3 DIMMs each of 16-bit interface. Also, the read latency is 8 clocks and write latency is 6 clocks. Thus with each clock corresponding to 1.87 ns for 533MHz clock

configuration parameter	gem5 default	modified for zedboard
device_bus_width	32	16
devices_per_rank	1	2
device_rowbuffer_size	1Kb	16Kb
tCL	15 ns	26.24 ns

Table 6.2: DRAM parameter changes

frequency (DDR3 clock domain), tCL is 26.24 ns. These parameters are derived from the configured hardware design using the xilinx platform studio as mentioned in section 4.2.1.

Bus width between L1 and L2 was changed to 8 bytes (64-bit) from 32 bytes in CacheConfig.py in configs-common directory. Number of cache ports were changed from 200 to 2 (one read and one write) in O3cpu.py in src/cpu/O3 directory.

6.5 Experimental observations

Initial configurations lead to an error of approximately 50% in the CPU execution cycle count reported by Gem5 compared to that measured on hardware. Performance counters configured to get information on nodispatch stalls, number of loads and stores and branch misses on the hardware revealed that almost 90% of the execution time was spent in nodispatch stalls which could be only due to 2 reasons: i)empty issue stage as the instruction side is waiting on an i-cache miss ii)waiting for loads and stores to complete to resolve RAW dependencies. Also, the number of load and store instructions exceeded the total number of integer instructions which means the major contributor to the execution time are the load and store instructions.

The huge error can then be only due to wrong configuration of load-store unit. Detailed analysis of the underlying memory system in Gem5 and its timing diagrams were studied. Finally, the assumption of wrong LS unit configuration was confirmed using the debug flag LSQUnit in the Gem5 simulation. It was observed that *upto* 4 loads and 4 stores were getting inserted into the load store queues in the *same* cpu tick as LQEntries and SQEntries were 4 each. On the real hardware, in a single cpu cycle either one load OR one store instruction can be executed and there is no load-store queue but there is a store buffer with 4 slots.

To mimic this load-store unit architecture in Gem5 we changed LQEntries to 1 from 4 and SQEntries remained at 4. This reduced the error drastically from almost 50% to less than 1% for the fixed point fft function which is an integer program for the fft size of 16.

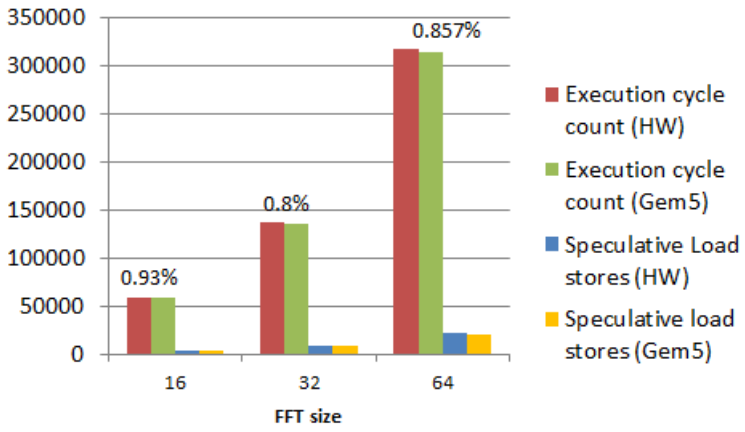


Figure 6.3: Execution Cycle count with error percentages on top of the graph

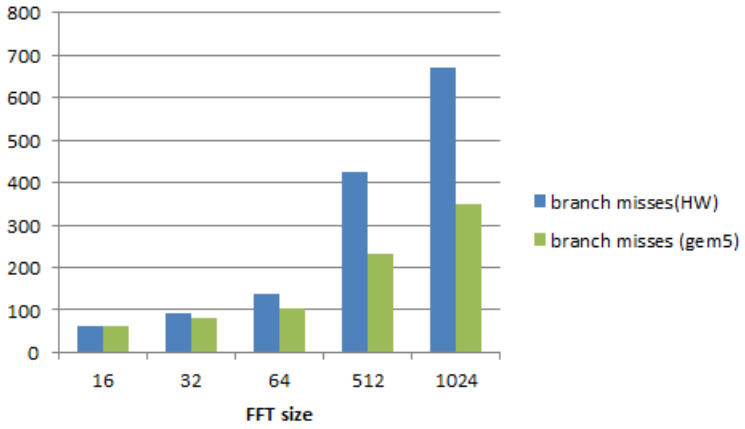


Figure 6.4: Branch misses in HW and Gem5

6.6 Results

Table 6.3 and graph 6.3 show the cpu execution cycle count observed on Gem5 and hardware for the fixed point fft function for varying fft sizes. The function operates only on integers. Table 6.4 and graph 6.4 show the branch mispredicts on Gem5 and hardware. The difference is due to different branch prediction schemes, tournament predictor on Gem5 and dynamic branch predictor on hardware leading to different speculative count of loads and stores. As tournament branch predictor is a better branch predictor leading to fewer loads and stores being executed, Gem5 is always faster than hardware except for the case when fft size is 1024.

Table 6.5 shows results for floating point programs for the same configuration. The two floating point programs chosen are the IIR-Gain-limiter algorithm used in Chapter 4 and a loop based program with a single floating point division operation in the body of the loop. The reason for high error rates are explained in the next section.

FFT size	Hardware execution cycle count	gem5 execution cycle count	Error percentage
16	58719	58172	0.93
32	137846	136013	0.8
64	317068	314348	0.857
512	3518095	3511870	0.1769
1024	7701365	7712243	-0.1412

Table 6.3: Execution cycle count for fixed point fft function

FFT size	HW branch mispredicts	speculative loads and stores	Gem5 branch mispredicts	speculative loads and stores
16	60	4125	62	3812
32	93	9649	80	9076
64	136	22054	104	21164
512	425	242380	233	239476
1024	671	530614	350	525948

Table 6.4: Speculative load stores and branch misses for fixed point fft

6.7 Floating Point errors

The high error percentages for floating point programs compared to integer programs are because the floating point unit and the load store unit are completely in-order in cortex-A9 whereas its completely out-of-order in Gem5. Consider the following piece of arm floating-point assembly code:

```
vdivs s15,s14,s15
vstr s15, [r3,#0]
ldr r3,[fp,#-8]
add r3,r3,#1
str r3,[fp,#-8]
```

On cortex-A9 the above piece of code would run completely in-order. However, in Gem5 we observed that the last store instruction is being issued, executed and committed before the

```
vstr s15, [r3,#0]
```

Program	sample/iter. size	HW cycle count	Gem5 cycle count	error percentage
IIR-Gain-Limiter	32	50842	45709	10
	64	98054	87213	11
	128	193116	169929	12
Float division	100	14232	9164	34

Table 6.5: Execution cycle count for IIR-Gain-Limiter and simple float division program

Program	Sample/iter. size	HW cycle count	Gem5 cycle count with manipulated LS queue entries	error percentage
IIR-Gain-Limiter	32	50842	50774	0.1
	64	98054	97166	0.9
	128	193116	189847	1.69
Float Division	100	14232	12896	9.3

Table 6.6: Execution cycle count for IIR-Gain-Limiter and simple float division program with manipulated load store queue entries

instruction. This is because of the long latency division instruction whose result the vstr instruction depends on. On cortex-A9 since all loads and stores are issued in-order, the vstr instruction must execute and complete before the next load and store instructions. However in Gem5, execution plus memory access latencies of *latest* loads and stores get hidden under latency of long division instructions before the *old* store instructions could complete write-back stage.

The solution is to force in-order load store by decreasing the store queue entries to 1 from 4. This yielded reduced errors for the simple float program and IIR-Gain-Limiter program as shown in table 6.6. This remedy is useful only if:

- There exists loops with long latency floating point and integer instructions such as the division operation.
- There are several load and store instructions around the long latency instructions.

The above solution should not be used for integer programs or region of interests without long latency operations. Typically, its hard to find load and store instructions in a short instruction window without any data dependencies on previous instructions. The execution cycle counts on HW and Gem5 with the correct and manipulated load store queue entries for the two floating point programs are represented in graph 6.5. The min and max error percentages corresponding to manipulated and correct load store queue entries are mentioned on top of the graphs.

6.8 Major differences between Gem5 and cortex-A9

The first major difference is that cortex-A9 is a mix between in-order and out-of-order resources and cpu models in Gem5 are completely out-of-order or in-order. Also, timing analysis on hardware was performed with xilinx zedboard and the machine type on Gem5 was the Realview PBX board though they both have the same cpu cores.

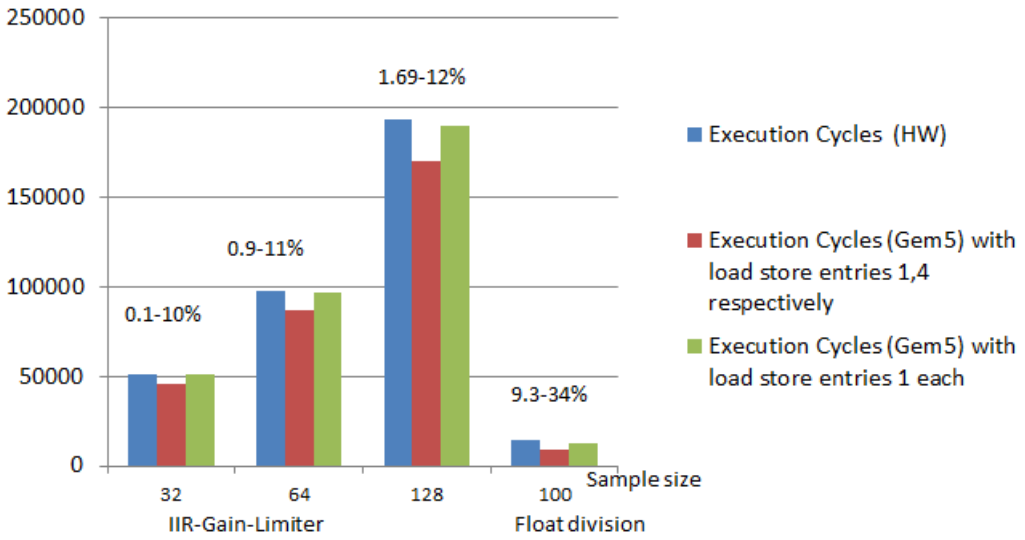


Figure 6.5: Execution cycle count on HW and Gem5 with min and max error percentages corresponding to manipulated and correct load store queue entries respectively mentioned on top of the graph

6.8.1 Memory system

Instruction and Data caches in the cortex-A9 perform critical word first filling of cache which means the word which caused the miss is fetched from higher levels of memory and sent to the cpu to continue with its operation and the rest of the bytes in the cache line fill up in the background. In Gem5, data and instruction transfers between caches or cache and main meory happen using packets which are 32 bytes (8 words) long. The transfer starts from the closest word-aligned address to the requested word. Thus, 32 bytes are transferred, updated in cache and cpu can read the requested data and resume with its operation. Gem5 does not allow you to configure size of each write buffers of the cache just the number of buffers. Hence, details such as size of each store buffer in L2 cache cannot be configured where as the number of buffers can be configured. The latencies of L2 cache vary with different scenarios such as hit-under-hit, hit-under-miss, miss-under-miss situations and hence one value for hit and response latencies might not be accurate. The classic memory system modelled in Gem5 is not 100% accurate. Static front-end and back-end latencies are used for the DRAM memory controller where as actual latencies in hardware depend on dynamic features such as arbitration policies. The load-store unit architecture is modelled slightly different than the one on actual hardware as discussed in subsection 6.5.

6.8.2 Branch prediction and Replacement policies

Gem5 models a tournament branch predictor and a dynamic branch predictor with only a global predictor is present in cortex-A9. In our experiment, though the different branch predictors did not contribute to the error significantly but it does contribute to

some error due to speculative execution. For example: Loads and stores are aborted if a branch is mispredicted and the pipeline is flushed hence significantly affecting execution time.

Caches (including TLB caches) on cortex-A9 employ pseudo-random/round-robin replacement policy and the cache replacement policy on Gem5 is LRU. In our experiment, no cache evictions took place due to capacity misses hence errors due to different replacement policies could not be evaluated.

6.9 Summary

Thus, execution time analysis on the cortex-A9 at the cycle-accurate abstraction level using a computer architecture simulator provided good accuracy levels for both floating-point and integer programs. Minor manipulations to the configurations were shown to increase accuracy levels further.

7 Conclusions

This chapter describes the main thesis contribution in section 7.1 which is the suggestion of a model/simulation framework to perform execution time analysis for both the single core and multi-core scenarios. Section 7.4 recommends future work that needs to be done for effective use of the model/simulation framework. The chapter is finally concluded with overall conclusions of the thesis in section 7.5.

7.1 Main thesis contribution

During the course of the thesis, few critical learnings about audio algorithms were made which helped us to suggest a model/simulation framework for performing execution time analysis for both the single core and multi-core scenarios.

Most audio algorithms are written using a single thread and have limited number of execution paths. Hence the control flow is fairly deterministic unlike control applications which can have multiple execution paths chosen depending on dynamically varying input parameters. Therefore, control flow in control applications changes non-deterministically during the runtime of the application. Also, control applications interact with the external environment and hence consist of several interrupts and communication with I/O peripherals. It is almost uncommon to find interrupts in an audio algorithm and the only peripheral they mainly interact with is DMA.

Some complex audio algorithms like adaptive filters, compressors and audio codecs consist of multiple execution paths depending on the value of the incoming audio sample. For example: For a speech input, the open source audio codec Opus, which performs encoding and decoding of speech or music, does not perform the encoding operation for parts of the input detected as silence. Any speech input has some parts of silence. Thus, depending on the value of input, different execution paths are chosen. This significantly influences execution time as different execution paths lead to different execution times. Therefore, inputs with shorter durations of silence lead to longer execution times compared to inputs with longer durations of silence.

To overcome the problem of varying execution times due to changing input values, static analysis is conducted to determine the worst case execution time irrespective of any input. There are hybrid approaches combining measurement and static analysis approaches to determine WCET like in [33] but there is no mention of the accuracy achieved. From section 5.4 in chapter 5 we can safely conclude that static analysis leads to huge overestimations. An alternative approach would be dynamic WCET using dynamic methods.

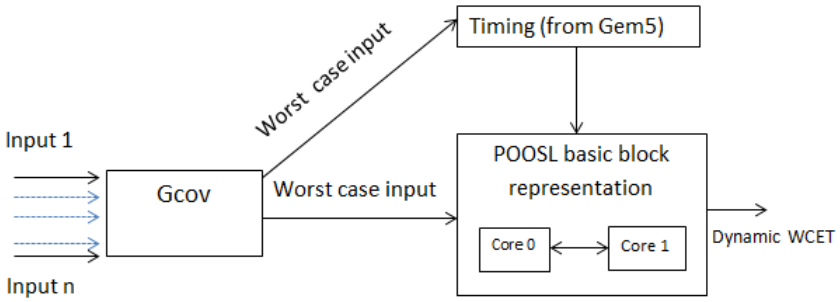


Figure 7.1: Model/simulation framework for dynamic WCET

Audio algorithms have hard real-time constraints. Hence, a worst case execution time is desired to be known. We suggest a simulation framework whose target is to provide a dynamic WCET estimate rather than a statically determined WCET. The framework comprises of three tools namely a coverage tool called Gcov, POOSL simulator and Gem5 simulator. Figure 7.1 represents the tool flow for the simulation framework for providing a dynamic WCET estimate.

Gcov is a code coverage tool from GNU tools which aims at providing input on how much of the code was actually executed during runtime. This is done in three phases:

- The code is compiled with gcc using special flags namely -fprofile-arcs and -ftest-coverage.
- The data collection dictated by the special flags takes place when the executable is run.
- The code coverage is finally reported to the user by running the gcov command.

The most common source of dynamism in audio algorithms affecting the WCET are the limited number of different execution paths which can be directly attributed to values of audio samples. Therefore, a code coverage tool like Gcov can be used to determine the worst case path when the algorithm is fed with a variety of inputs. The input which triggers the worst case path leads to maximum code coverage and hence can be chosen for further analysis. The execution contexts 'captured' in this worst case path are the only relevant contexts for us which will be executed in Gem5 and POOSL. Thus, the problem of execution contexts of basic blocks is solved in an implicit way and no additional analysis for context-aware basic block timing is required.

For the single core scenario, if a single algorithm on a single core needs to be analysed for execution time, Gem5 would be sufficient. Also, if hardware is available, cycle counters on hardware can provide execution cycle count of the algorithm.

However, for the multi-core scenario, two cores can communicate with each other either through asynchronous software generated interrupts or in a synchronous fashion using shared memory. Porting problems associated with porting multi-core software on

hardware can effectively delay execution time analysis if hardware is used. Gem5 is modelled very close to hardware and these porting problems will be apparent on Gem5 as well. Abstraction comes to our rescue here. Correct initializations of interrupt controllers, shared memory and other porting problems can be alleviated by using POOSL for inter-core communication.

POOSL process classes are used to represent different cores since processes in POOSL are executed in parallel. The algorithms on all the cores are specified at the basic block level due to the advantages of this abstraction level discussed in Chapter 5. The interrupt primitive in POOSL can be used to mimic the software generated interrupts between the cores and synchronous message-pairs between processes can mimic the shared memory communication between cores. The worst case input can be fed into the POOSL model to *simulate* the worst case path. Timing values in terms of execution cycles of basic blocks are provided by Gem5. The POOSL model thus created when simulated yields the dynamic WCET estimate for the multi-core in AMP mode scenario for audio algorithms.

7.2 Comparison with related work

[34] proposes a cycle accurate instruction set simulator by combining QEMU (Quick Emulator) which is a hardware emulator used for functional simulation and System-C, a C++ library used for low-level hardware modelling and timing simulation. The author's motivation is to be able to calculate performance estimates using hardware/software co-simulation for SoC platforms. In this approach QEMU is used to extract information about instructions executed and data accesses. This information is passed to System-C for timing simulation as QEMU does not support any concept of time. Certain dynamic behaviours such as branch-prediction, hardware pipeline, out-of-order processing, super-scalar pipelines and multi-level cache behaviour have to be modelled in System-C as QEMU can only be used to extract information about addresses of instructions executed and register file accesses. Thus, the approach in [34] is unsuitable for complex micro-architectures as exhibited by processors like Cortex-A9 which was the focus of our thesis. Moreover, the approach in [34] is validated with ARM9 processor which is a simple 5-stage pipeline processor and caches have not been considered. We choose Cortex-A9 for validation.

[17] proposes an annotated System-C model which combines statically calculated cycle counts of basic blocks of a program with corrections due to pipeline overlap, branch misprediction and cache misses using branch prediction, pipeline overlap analysis and cache models written in System-C. This methodology fails for indirect addressing modes which depend on the contents of registers that can be determined only by executing the program. We solve this by *simulating* the execution of the instructions using Gem5 at the microarchitecture level which accounts for pipeline overlap, dynamic cache accesses using indirect addressing and branch-prediction. Validation of the approach in [17] is done using a Tricore processor from Infineon which lacks complex microarchitectural features like out-of-order processing and branch prediction present in Cortex-A9.

In [33], an approach which combines measurement based techniques with static anal-

Method	Negative aspects	Thesis features
QEMU + SystemC [34]	1. Unsuitable for complex microarchitectures. 2. Validated with simple processors.	1. Gem5 aims at complex microarchitectures. 2. Validated with Cortex-A9
Annotated SystemC [17]	1. Static cycle counts of basic blocks. 2. Requires detailed Branch prediction and cache models in SystemC. 3. Static pipeline overlap analysis involved. 4. Fails to analyse indirect cache accesses which requires execution of the code. 5. Unsuitable for complex microarchitectures. 6. Validated with simple processors	1. Basic block cycle counts determined dynamically. 2. Gem5 simulates all cache accesses, branch predictions and pipeline overlap cycle-accurately. 3. Validated with Cortex-A9.
Hybrid methods [33]	1. Control flow graph analysis is static and requires user annotations. 2. WCET calculation is done using static means.	1. Longest execution path is simulated and requires no user intervention. 2. WCET is determined through simulation.

Table 7.1: Comparison of thesis with related work

yses to determine WCET (worst case execution time). The approach involves static control flow graph analysis to aid in test input generation for WCET determination. Hence requires user annotations for loop bounds. The test inputs generated are then used for execution of the program and measurements of basic block execution cycles count. These measurements are finally combined with static analysis techniques like Implicit path enumeration technique to determine WCET. Since in our experiments we observed huge overestimations using static analysis techniques for execution time estimation, we focus on dynamic WCET estimation methods. In this thesis, the longest execution path is *simulated* and not statically determined and hence no user intervention is required.

Table 7.1 provides a summary of the above discussion targeting execution time analysis. The table also explains the shortcomings of these methods and how the model/simulation framework overcomes these shortcomings.

7.3 Critical analysis

The model/simulation framework suggested in this thesis works only if the C code of the audio algorithm whose execution time has to be determined is available. Also, the C code

of the control functionality which interrupts the audio processing should be available. The framework assumes the audio engineer has good knowledge of the audio algorithm and is able to determine the worst case input by inspecting the executed count of basic blocks shown by Gcov tool. Both the audio algorithm and the control functionality can be expressed at the algorithmic abstraction level or the basic block abstraction level depending on the flexibility of the model desired.

The *assembly* code of the audio algorithm should be instrumented with the reset stats feature of Gem5 to obtain the execution cycle counts of basic blocks. Combined correction factor for pipeline overlap and dynamic data accesses between iterations of a loop have to be measured in Gem5 and fed in as correction factors for each basic block in the POOSL model. This can be done by taking the difference in execution cycle count of loop body (basic block) for one iteration and N iterations and calculating average error per iteration comprising of both pipeline overlap and dynamic data cache hit/miss latencies. It is important that the number of instructions in each basic block is greater or equal to the instruction window size of the processor as modern processors execute the instructions within the instruction window out-of-order if no true data dependencies exist. Therefore, the execution time measurement of each basic block on Gem5 includes out-of-order execution effects.

A complex audio algorithm consists of several function calls to smaller algorithms. When several algorithm blocks containing basic blocks (modelled as POOSL processes or methods) are combined to model a more complex algorithm, care should be taken to represent context switches (between function calls) as delays. Hence there must be a black box implementation bridging algorithm blocks together. These black boxes should contain appropriate delays representing context save/restore and any other bridging code.

7.4 Future work

For the simulation framework described in section 7.1 to be useful, a minimalistic library of audio algorithms required by public address and conference systems at Bosch Security Systems should be created such that any complex algorithm can be built connecting different code blocks in this library. Also, a library containing a large variety of audio sample inputs is required to test code coverage and determine the worst case path. Once these libraries are created, they can be specified and modelled using POOSL, and Gem5 can be used to get execution cycle counts of all basic blocks by using the checkpointing and resetting statistics features of Gem5. However, if more accuracy is desired, features such as bus arbitration policies when the two cores compete for the bus to access shared memory might have to be modelled. The whole exercise is a one time effort and once the flexible POOSL models are created, execution time analysis of any complex algorithm can be performed.

On the Gem5 side, the target hardware and the Gem5 models can be made more similar if the branch prediction policies, cache behaviour, cache replacement policies and prefetching policies in Gem5 are modelled exactly as it appears in the hardware. This could lead to almost 100% accuracy.

7.5 Overall Conclusions

From the various experiments conducted during this thesis, it can be safely concluded that hardware measurements are most accurate. This method is however not flexible and not suitable for architectural exploration i.e design space exploration of different processors. For execution time analysis on multi-core processors, porting problems associated with multi-core software is a big hurdle to cross before execution time analysis can be done if hardware is used. Static analysis for such complex processors leads to high levels of inaccuracy as the processors consist of a variety of dynamic features. Thus, the model/simulation framework suggested in this thesis provides a suitable solution for execution time analysis even for challenging multi-core scenarios.

Bibliography

[1] J. E. Andreas Ermedahl, "Execution time analysis for embedded real-time systems."

[2] `https://ti.tuwien.ac.at/cps/teaching/courses/wcet/slides/wcet05_hw_modeling_3.pdf/`.

[3] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and synchronization*. CRC press, 2009.

[4] B. Theelen, P. van der Putten, and J. Voeten, "Using the she method for uml-based performance modelling," 2003.

[5] R. Domer, "Transaction level modeling of computation," tech. rep., University of California, Irvine, 2006.

[6] `www.arm.com/files/pdf/CortexM3_Uni_Intro.pdf`.

[7] `www.gem5.org/dist/tutorials/hipeac2012/gem5_hipeac.pdf`.

[8] `www.arm.com/files/pdf/armcortexa-9processors.pdf`.

[9] G. Davis and R. Jones, *Sound Reinforcement Handbook*.

[10] J. Engblom, *Processor pipelines and static worst-case execution time analysis*. PhD thesis, Uppsala university, April 2002.

[11] E. Lee and D. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.

[12] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 63–72, 2003.

[13] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis,"

[14] A. E. Gozek, "Task execution time prediction for motion control applications," Master's thesis, TU Eindhoven, 2013.

[15] K.-L. Lin, P.-J. Lin, C.-K. Lo, and R.-S. Tsay, "Fast and accurate tlm computation model generation using source-level timing annotation,"

[16] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," in *ACM SIGARCH Computer Architecture News*, vol. 25, pp. 13–25, 1997.

[17] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Proceedings of the 45th annual Design Automation Conference*, 2008.

[18] "The multi2sim simulation framework."

[19] `http://pages.cs.wisc.edu/~markhill/DineroIV/`.

[20] V. M. Weaver, *Using Dynamic Binary Instrumentation to create faster, validated, multi-core simulations*. PhD thesis, Cornell University, May 2010.

[21] I. software limited., "Ovp guide to using processor models."

[22] Altera, "Soc fpga arm cortex-a9 mpcore processor advance information brief."

[23] `http://www.carbondesignsystems.com/`.

[24] `http://www.synopsys.com/Community/Interoperability/SystemLevelCatalyst/Pages/MVaST.aspx`.

[25] CoWare, "Coware processor designer."

[26] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0255m/Cacjgfad.html`.

[27] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 290–295, IEEE, 2008.

[28] R. Plyaskin, *Fast and accurate performance simulation of out-of-order processing cores in embedded systems*. PhD thesis, Technical universitat Munich, June 2014.

[29] N. Binkert, B. Beckmann, G. Black, A. Saidi, and et al, "The gem5 simulator," in *ACM SIGARCH Computer Architecture News*, vol. 39(2), pp. 1–7, 2011.

[30] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *International Symposium on High Performance Computer Architecture*, HPCA'19, 2013.

[31] V. Spiliopoulos, A. Bagdia, A. Hansson, P. Aldworth, and S. Kaxiras, "Introducing dvfs-management in a full-system simulator," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, pp. 535–545, Aug 2013.

[32] `http://pastebin.com/t1AU4D7H`.

[33] R. Kirner, P. Puschner, and I. Wenzel, "Measurement-based worst-case execution time analysis using automatic test-data generation," in *IN PROC. IEEE WORKSHOP ON SOFTWARE TECH. FOR FUTURE EMBEDDED AND UBIQUITOUS SYSTS. (SEUS05*, pp. 7–10, 2004.

[34] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A qemu and systemc-based cycle-accurate iss for performance estimation on soc development," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, pp. 593–606, April 2011.

8 Appendix

8.1 Appendix A

```
/*      fix_fft.c - Fixed-point Fast Fourier Transform  */
/*
        fix_fft()       perform FFT or inverse FFT
        window()        applies a Hanning window to the (time) input
        fix_loud()      calculates the loudness of the signal, for
                        each freq point. Result is an integer array,
                        units are dB (values will be negative).
        iscale()        scale an integer value by (numer/denom).
        fix_mpy()       perform fixed-point multiplication.
        Sinewave[1024]  sinewave normalized to 32767 (= 1.0).
        Loudampl[100]   Amplitudes for lopudnesses from 0 to -99 dB.
        Low_pass        Low-pass filter, cutoff at sample_freq / 4.


        All data are fixed-point short integers, in which
        -32768 to +32768 represent -1.0 to +1.0. Integer arithmetic
        is used for speed, instead of the more natural floating-point.

        For the forward FFT (time -> freq), fixed scaling is
        performed to prevent arithmetic overflow, and to map a 0dB
        sine/cosine wave (i.e. amplitude = 32767) to two -6dB freq
        coefficients; the one in the lower half is reported as 0dB
        by fix_loud(). The return value is always 0.

        For the inverse FFT (freq -> time), fixed scaling cannot be
        done, as two 0dB coefficients would sum to a peak amplitude of
        64K, overflowing the 32k range of the fixed-point integers.
        Thus, the fix_fft() routine performs variable scaling, and
        returns a value which is the number of bits LEFT by which
        the output must be shifted to get the actual amplitude
        (i.e. if fix_fft() returns 3, each value of fr[] and fi[]
        must be multiplied by 8 (2**3) for proper scaling.
        Clearly, this cannot be done within the fixed-point short
        integers. In practice, if the result is to be used as a
        filter, the scale_shift can usually be ignored, as the
        result will be approximately correctly normalized as is.


        TURBO C, any memory model; uses inline assembly for speed
        and for carefully-scaled arithmetic.

        Written by:  Tom Roberts  11/8/89
        Made portable:  Malcolm Slaney 12/15/94 malcolm@interval.com
```

```
                Timing on a Macintosh PowerBook 180.... (using Symantec C6
    .0)
                        fix_fft (1024 points)              8 ticks
                        fft (1024 points - Using SANE)   112 Ticks
                        fft (1024 points - Using FPU)     11

*/

/* FIX_MPY() - fixed-point multiplication macro.
   This macro is a statement, not an expression (uses asm).
   BEWARE: make sure _DX is not clobbered by evaluating (A) or DEST.
   args are all of type fixed.
   Scaling ensures that 32767*32767 = 32767. */



#include        <math.h>

#define M       4
#define N       (1<<M)

#define dosFIX_MPY(DEST,A,B)        {       \
        _DX = (B);                          \
        _AX = (A);                          \
        asm imul dx;                        \
        asm add ax,ax;                      \
        asm adc dx,dx;                      \
        DEST = _DX;             }

#define FIX_MPY(DEST,A,B)       DEST = ((long)(A) * (long)(B))>>15

#define N_WAVE          1024    /* dimension of Sinewave[] */
#define LOG2_N_WAVE     10      /* log2(N_WAVE) */
#define N_LOUD          100     /* dimension of Loudampl[] */
#ifndef fixed
#define fixed short
#endif

#if N_WAVE != 1024
        ERROR: N_WAVE != 1024
#endif
fixed Sinewave[1024] = {
      0,    201,    402,    603,    804,   1005,   1206,   1406,
   1607,   1808,   2009,   2209,   2410,   2610,   2811,   3011,
   3211,   3411,   3611,   3811,   4011,   4210,   4409,   4608,
   4807,   5006,   5205,   5403,   5601,   5799,   5997,   6195,
   6392,   6589,   6786,   6982,   7179,   7375,   7571,   7766,
   7961,   8156,   8351,   8545,   8739,   8932,   9126,   9319,
   9511,   9703,   9895,  10087,  10278,  10469,  10659,  10849,
  11038,  11227,  11416,  11604,  11792,  11980,  12166,  12353,
  12539,  12724,  12909,  13094,  13278,  13462,  13645,  13827,
  14009,  14191,  14372,  14552,  14732,  14911,  15090,  15268,
```

```
15446,  15623,  15799,  15975,  16150,  16325,  16499,  16672,
16845,  17017,  17189,  17360,  17530,  17699,  17868,  18036,
18204,  18371,  18537,  18702,  18867,  19031,  19194,  19357,
19519,  19680,  19840,  20000,  20159,  20317,  20474,  20631,
20787,  20942,  21096,  21249,  21402,  21554,  21705,  21855,
22004,  22153,  22301,  22448,  22594,  22739,  22883,  23027,
23169,  23311,  23452,  23592,  23731,  23869,  24006,  24143,
24278,  24413,  24546,  24679,  24811,  24942,  25072,  25201,
25329,  25456,  25582,  25707,  25831,  25954,  26077,  26198,
26318,  26437,  26556,  26673,  26789,  26905,  27019,  27132,
27244,  27355,  27466,  27575,  27683,  27790,  27896,  28001,
28105,  28208,  28309,  28410,  28510,  28608,  28706,  28802,
28897,  28992,  29085,  29177,  29268,  29358,  29446,  29534,
29621,  29706,  29790,  29873,  29955,  30036,  30116,  30195,
30272,  30349,  30424,  30498,  30571,  30643,  30713,  30783,
30851,  30918,  30984,  31049,
31113,  31175,  31236,  31297,
31356,  31413,  31470,  31525,  31580,  31633,  31684,  31735,
31785,  31833,  31880,  31926,  31970,  32014,  32056,  32097,
32137,  32176,  32213,  32249,  32284,  32318,  32350,  32382,
32412,  32441,  32468,  32495,  32520,  32544,  32567,  32588,
32609,  32628,  32646,  32662,  32678,  32692,  32705,  32717,
32727,  32736,  32744,  32751,  32757,  32761,  32764,  32766,
32767,  32766,  32764,  32761,  32757,  32751,  32744,  32736,
32727,  32717,  32705,  32692,  32678,  32662,  32646,  32628,
32609,  32588,  32567,  32544,  32520,  32495,  32468,  32441,
32412,  32382,  32350,  32318,  32284,  32249,  32213,  32176,
32137,  32097,  32056,  32014,  31970,  31926,  31880,  31833,
31785,  31735,  31684,  31633,  31580,  31525,  31470,  31413,
31356,  31297,  31236,  31175,  31113,  31049,  30984,  30918,
30851,  30783,  30713,  30643,  30571,  30498,  30424,  30349,
30272,  30195,  30116,  30036,  29955,  29873,  29790,  29706,
29621,  29534,  29446,  29358,  29268,  29177,  29085,  28992,
28897,  28802,  28706,  28608,  28510,  28410,  28309,  28208,
28105,  28001,  27896,  27790,  27683,  27575,  27466,  27355,
27244,  27132,  27019,  26905,  26789,  26673,  26556,  26437,
26318,  26198,  26077,  25954,  25831,  25707,  25582,  25456,
25329,  25201,  25072,  24942,  24811,  24679,  24546,  24413,
24278,  24143,  24006,  23869,  23731,  23592,  23452,  23311,
23169,  23027,  22883,  22739,  22594,  22448,  22301,  22153,
22004,  21855,  21705,  21554,  21402,  21249,  21096,  20942,
20787,  20631,  20474,  20317,  20159,  20000,  19840,  19680,
19519,  19357,  19194,  19031,  18867,  18702,  18537,  18371,
18204,  18036,  17868,  17699,  17530,  17360,  17189,  17017,
16845,  16672,  16499,  16325,  16150,  15975,  15799,  15623,
15446,  15268,  15090,  14911,  14732,  14552,  14372,  14191,
14009,  13827,  13645,  13462,  13278,  13094,  12909,  12724,
12539,  12353,  12166,  11980,  11792,  11604,  11416,  11227,
11038,  10849,  10659,  10469,  10278,  10087,   9895,   9703,
 9511,   9319,   9126,   8932,   8739,   8545,   8351,   8156,
 7961,   7766,   7571,   7375,   7179,   6982,   6786,   6589,
 6392,   6195,   5997,   5799,   5601,   5403,   5205,   5006,
 4807,   4608,   4409,   4210,   4011,   3811,   3611,   3411,
 3211,   3011,   2811,   2610,   2410,   2209,   2009,   1808,
 1607,   1406,   1206,   1005,    804,    603,    402,    201,
```

```
    0,   −201,   −402,   −603,   −804,  −1005,  −1206,  −1406,
−1607,  −1808,  −2009,  −2209,  −2410,  −2610,  −2811,  −3011,
−3211,  −3411,  −3611,  −3811,  −4011,  −4210,  −4409,  −4608,
−4807,  −5006,  −5205,  −5403,  −5601,  −5799,  −5997,  −6195,
−6392,  −6589,  −6786,  −6982,  −7179,  −7375,  −7571,  −7766,
−7961,  −8156,  −8351,  −8545,  −8739,  −8932,  −9126,  −9319,
−9511,  −9703,  −9895, −10087, −10278, −10469, −10659, −10849,
−11038, −11227, −11416, −11604, −11792, −11980, −12166, −12353,
−12539, −12724, −12909, −13094, −13278, −13462, −13645, −13827,
−14009, −14191, −14372, −14552, −14732, −14911, −15090, −15268,
−15446, −15623, −15799, −15975, −16150, −16325, −16499, −16672,
−16845, −17017, −17189, −17360, −17530, −17699, −17868, −18036,
−18204, −18371, −18537, −18702, −18867, −19031, −19194, −19357,
−19519, −19680, −19840, −20000, −20159, −20317, −20474, −20631,
−20787, −20942, −21096, −21249, −21402, −21554, −21705, −21855,
−22004, −22153, −22301, −22448, −22594, −22739, −22883, −23027,
−23169, −23311, −23452, −23592, −23731, −23869, −24006, −24143,
−24278, −24413, −24546, −24679, −24811, −24942, −25072, −25201,
−25329, −25456, −25582, −25707, −25831, −25954, −26077, −26198,
−26318, −26437, −26556, −26673, −26789, −26905, −27019, −27132,
−27244, −27355, −27466, −27575, −27683, −27790, −27896, −28001,
−28105, −28208, −28309, −28410, −28510, −28608, −28706, −28802,
−28897, −28992, −29085, −29177, −29268, −29358, −29446, −29534,
−29621, −29706, −29790, −29873, −29955, −30036, −30116, −30195,
−30272, −30349, −30424, −30498, −30571, −30643, −30713, −30783,
−30851, −30918, −30984, −31049, −31113, −31175, −31236, −31297,
−31356, −31413, −31470, −31525, −31580, −31633, −31684, −31735,
−31785, −31833, −31880, −31926, −31970, −32014, −32056, −32097,
−32137, −32176, −32213, −32249, −32284, −32318, −32350, −32382,
−32412, −32441, −32468, −32495, −32520, −32544, −32567, −32588,
−32609, −32628, −32646, −32662, −32678, −32692, −32705, −32717,
−32727, −32736, −32744, −32751, −32757, −32761, −32764, −32766,
−32767, −32766, −32764, −32761, −32757, −32751, −32744, −32736,
−32727, −32717, −32705, −32692, −32678, −32662, −32646, −32628,
−32609, −32588, −32567, −32544, −32520, −32495, −32468, −32441,
−32412, −32382, −32350, −32318, −32284, −32249, −32213, −32176,
−32137, −32097, −32056, −32014, −31970, −31926, −31880, −31833,
−31785, −31735, −31684, −31633, −31580, −31525, −31470, −31413,
−31356, −31297, −31236, −31175, −31113, −31049, −30984, −30918,
−30851, −30783, −30713, −30643, −30571, −30498, −30424, −30349,
−30272, −30195, −30116, −30036, −29955, −29873, −29790, −29706,
−29621, −29534, −29446, −29358, −29268, −29177, −29085, −28992,
−28897, −28802, −28706, −28608, −28510, −28410, −28309, −28208,
−28105, −28001, −27896, −27790, −27683, −27575, −27466, −27355,
−27244, −27132, −27019, −26905, −26789, −26673, −26556, −26437,
−26318, −26198, −26077, −25954, −25831, −25707, −25582, −25456,
−25329, −25201, −25072, −24942, −24811, −24679, −24546, −24413,
−24278, −24143, −24006, −23869, −23731, −23592, −23452, −23311,
−23169, −23027, −22883, −22739, −22594, −22448, −22301, −22153,
−22004, −21855, −21705, −21554, −21402, −21249, −21096, −20942,
−20787, −20631, −20474, −20317, −20159, −20000, −19840, −19680,
−19519, −19357, −19194, −19031, −18867, −18702, −18537, −18371,
−18204, −18036, −17868, −17699, −17530, −17360, −17189, −17017,
−16845, −16672, −16499, −16325, −16150, −15975, −15799, −15623,
−15446, −15268, −15090, −14911, −14732, −14552, −14372, −14191,
```

```
 -14009, -13827, -13645, -13462, -13278, -13094, -12909, -12724,
 -12539, -12353, -12166, -11980, -11792, -11604, -11416, -11227,
 -11038, -10849, -10659, -10469, -10278, -10087,  -9895,  -9703,
  -9511,  -9319,  -9126,  -8932,  -8739,  -8545,  -8351,  -8156,
  -7961,  -7766,  -7571,  -7375,  -7179,  -6982,  -6786,  -6589,
  -6392,  -6195,  -5997,  -5799,  -5601,  -5403,  -5205,  -5006,
  -4807,  -4608,  -4409,  -4210,  -4011,  -3811,  -3611,  -3411,
  -3211,  -3011,  -2811,  -2610,  -2410,  -2209,  -2009,  -1808,
  -1607,  -1406,  -1206,  -1005,   -804,   -603,   -402,   -201,
};
 /* placed at end of this file for clarity */

int fix_fft(fixed fr[], fixed fi[], int m, int inverse);
fixed fix_mpy(fixed a, fixed b);

/*
        fix_fft() - perform fast Fourier transform.

        if n>0 FFT is done, if n<0 inverse FFT is done
        fr[n],fi[n] are real,imaginary arrays, INPUT AND RESULT.
        size of data = 2**m
        set inverse to 0=dft, 1=idft
*/
int fix_fft(fixed fr[], fixed fi[], int m, int inverse)
{
        int mr,nn,i,j,l,k,istep, n, scale, shift;
        fixed qr,qi,tr,ti,wr,wi;

                n = 1<<m;

        if(n > N_WAVE)
                return -1;

        mr = 0;
        nn = n - 1;
        scale = 0;

        /* decimation in time - re-order data */
        for(m=1; m<=nn; ++m) {
                l = n;
                do {
                        l >>= 1;
                } while(mr+l > nn);
                mr = (mr & (l-1)) + l;

                if(mr <= m) continue;
                tr = fr[m];
                fr[m] = fr[mr];
                fr[mr] = tr;
                ti = fi[m];
                fi[m] = fi[mr];
                fi[mr] = ti;
        }

        l = 1;
```

```
        k = LOG2_N_WAVE-1;
        while(l < n) {
                if(inverse) {
                        /* variable scaling, depending upon data */
                        shift = 0;
                        for(i=0; i<n; ++i) {
                                j = fr[i];
                                if(j < 0)
                                        j = -j;
                                m = fi[i];
                                if(m < 0)
                                        m = -m;
                                if(j > 16383 || m > 16383) {
                                        shift = 1;
                                        break;
                                }
                        }
                        if(shift)
                                ++scale;
                } else {
                        /* fixed scaling, for proper normalization -
                           there will be log2(n) passes, so this
                           results in an overall factor of 1/n,
                           distributed to maximize arithmetic accuracy. */
                        shift = 1;
                }
                /* it may not be obvious, but the shift will be performed
                   on each data point exactly once, during this pass. */
                istep = l << 1;
                for(m=0; m<l; ++m) {
                        j = m << k;
                        /* 0 <= j < N_WAVE/2 */
                        wr =  Sinewave[j+N_WAVE/4];
                        wi = -Sinewave[j];
                        if(inverse)
                                wi = -wi;
                        if(shift) {
                                wr >>= 1;
                                wi >>= 1;
                        }
                        for(i=m; i<n; i+=istep) {
                                j = i + l;
                                        tr = fix_mpy(wr,fr[j]) -
fix_mpy(wi,fi[j]);
                                        ti = fix_mpy(wr,fi[j]) +
fix_mpy(wi,fr[j]);
                                qr = fr[i];
                                qi = fi[i];
                                if(shift) {
                                        qr >>= 1;
                                        qi >>= 1;
                                }
                                fr[j] = qr - tr;
                                fi[j] = qi - ti;
                                fr[i] = qr + tr;
```

```
                                fi[i] = qi + ti;
                        }
                }
                --k;
                l = istep;
        }

        return scale;
}

/*
        fix_mpy() - fixed-point multiplication
*/
fixed fix_mpy(fixed a, fixed b)
{
        FIX_MPY(a,a,b);
        return a;
}

int main(){
        fixed real[N], imag[N];
        int     i;

        for (i=0; i<N; i++){
                real[i] = 1000*cos(i*2*3.1415926535/N);
                imag[i] = 0;
        }

        fix_fft(real, imag, M, 0);

        return(0);
}
```

8.2 Appendix B

```
# Copyright (c) 2012 The Regents of The University of Michigan
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are
# met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution;
# neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
```

```
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
# Authors: Ron Dreslinski


from m5.objects import *

# Simple ALU Instructions have a latency of 1
class Exynos_Simple_Int(FUDesc):
    opList = [ OpDesc(opClass='IntAlu', opLat=1) ]
    count = 1

# Complex ALU instructions have a variable latencies
class Exynos_Complex_Int(FUDesc):
    opList = [ OpDesc(opClass='IntMult', opLat=4, issueLat=1),
               OpDesc(opClass='IntAlu', opLat=1),
               OpDesc(opClass='IntDiv', opLat=12, issueLat=12),
               OpDesc(opClass='IprAccess', opLat=3, issueLat=1) ]
    count = 1


# Floating point and SIMD instructions
class Exynos_FP(FUDesc):
    opList = [ OpDesc(opClass='SimdAdd', opLat=4),
               OpDesc(opClass='SimdAddAcc', opLat=4),
               OpDesc(opClass='SimdAlu', opLat=4),
               OpDesc(opClass='SimdCmp', opLat=4),
               OpDesc(opClass='SimdCvt', opLat=3),
               OpDesc(opClass='SimdMisc', opLat=3),
               OpDesc(opClass='SimdMult',opLat=5),
               OpDesc(opClass='SimdMultAcc',opLat=5),
               OpDesc(opClass='SimdShift',opLat=3),
               OpDesc(opClass='SimdShiftAcc', opLat=3),
               OpDesc(opClass='SimdSqrt', opLat=9),
               OpDesc(opClass='SimdFloatAdd',opLat=5),
               OpDesc(opClass='SimdFloatAlu',opLat=5),
               OpDesc(opClass='SimdFloatCmp', opLat=3),
               OpDesc(opClass='SimdFloatCvt', opLat=3),
               OpDesc(opClass='SimdFloatDiv', opLat=3),
               OpDesc(opClass='SimdFloatMisc', opLat=3),
               OpDesc(opClass='SimdFloatMult', opLat=3),
               OpDesc(opClass='SimdFloatMultAcc',opLat=1),
               OpDesc(opClass='SimdFloatSqrt', opLat=9),
               OpDesc(opClass='FloatAdd', opLat=5),
               OpDesc(opClass='FloatCmp', opLat=5),
               OpDesc(opClass='FloatCvt', opLat=5),
               OpDesc(opClass='FloatDiv', opLat=9, issueLat=9),
               OpDesc(opClass='FloatSqrt', opLat=33, issueLat=33),
               OpDesc(opClass='FloatMult', opLat=4) ]
```

```
    count = 1


# Load/Store Units
class Exynos_LS(FUDesc):
    opList = [ OpDesc(opClass='MemRead',opLat=2),
               OpDesc(opClass='MemWrite',opLat=2) ]
    count = 1


# Functional Units for this CPU
class Exynos_FUP(FUPool):
    FUList = [Exynos_Simple_Int(), Exynos_Complex_Int(),
              Exynos_LS(), Exynos_FP()]

# Tournament Branch Predictor
class Exynos_BP(BranchPredictor):
    predType = "tournament"
    localPredictorSize = 512
    localCtrBits = 2
    localHistoryTableSize = 512
    globalPredictorSize = 2048
    globalCtrBits = 2
    choicePredictorSize = 2048
    choiceCtrBits = 2
    BTBEntries = 512
    BTBTagSize = 18
    RASSize = 8
    instShiftAmt = 2

class Exynos_3(DerivO3CPU):
    LQEntries = 4
    SQEntries = 4
    LSQDepCheckShift = 0
    LFSTSize = 1024
    SSITSize = 1024
    decodeToFetchDelay = 1
    renameToFetchDelay = 1
    iewToFetchDelay = 1
    commitToFetchDelay = 1
    renameToDecodeDelay = 1
    iewToDecodeDelay = 1
    commitToDecodeDelay = 1
    iewToRenameDelay = 1
    commitToRenameDelay = 1
    commitToIEWDelay = 1
    fetchWidth = 2
    fetchBufferSize = 16
    fetchToDecodeDelay = 3
    decodeWidth = 2
    decodeToRenameDelay = 2
    renameWidth = 2
    renameToIEWDelay = 1
    issueToExecuteDelay = 1
    dispatchWidth = 2
```

```
        issueWidth = 2
        wbWidth = 1
        wbDepth = 1
        fuPool = Exynos_FUP()
        iewToCommitDelay = 1
        renameToROBDelay = 1
        commitWidth = 2
        squashWidth = 2
        trapLatency = 37
        backComSize = 5
        forwardComSize = 5
        numPhysIntRegs = 56
        numPhysFloatRegs = 192
        numIQEntries = 16
        numROBEntries = 40

        switched_out = False
        branchPred = Exynos_BP()

# Instruction Cache
class Exynos_ICache(BaseCache):
        hit_latency = 4
        response_latency = 4
        mshrs = 2
        tgts_per_mshr = 8
        size = '32kB'
        assoc = 4
        is_top_level = 'true'

# Data Cache
class Exynos_DCache(BaseCache):
        hit_latency = 4
        response_latency = 4
        mshrs = 6
        tgts_per_mshr = 8
        size = '32kB'
        assoc = 4
        write_buffers = 16
        is_top_level = 'true'

# TLB Cache
# Use a cache as a L2 TLB
class ExynosWalkCache(BaseCache):
        hit_latency = 7
        response_latency = 7
        mshrs = 6
        tgts_per_mshr = 8
        size = '2kB'
        assoc = 2
        write_buffers = 16
        is_top_level = 'true'


# L2 Cache
class ExynosL2(BaseCache):
```

```
    hit_latency = 37
    response_latency = 37
    mshrs = 16
    tgts_per_mshr = 8
    size = '1MB'
    assoc = 16
    write_buffers = 8
    prefetch_on_access = 'true'
    # Simple stride prefetcher
    prefetcher = StridePrefetcher(degree=8, latency = 3)
```