# EARLY AND ACCURATE MODELING OF STREAMING

## **EMBEDDED APPLICATIONS**

## **RICHARD LEE**

A Thesis

in

The Department Of Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science (Electrical and Computer Engineering)

at

Concordia University Montréal, Québec, Canada

August 2013

© RICHARD LEE, 2013

### CONCORDIA UNIVERSITY SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Richard Lee

Entitled: "Early and Accurate Modeling of Streaming Embedded Applications"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

	Chair
Dr. Zahangir Kabir	
	Examiner, External
Dr. Benjamin Fung	To the Program
	Examiner
Dr. Abdelwahab Hamou-Lhadj	
	Supervisor
Dr. Samar Abdi	

Approved by: Dr. W. E. Lynch, Chair Department of Electrical and Computer Engineering

August 2013

Dr. Christopher Trueman Dean, Faculty of Engineering and Computer Science

## ABSTRACT

#### EARLY AND ACCURATE MODELING OF STREAMING EMBEDDED APPLICATIONS

#### **Richard Lee**

This thesis presents automatic generation of fast and accurate timed models of streaming embedded applications, before the complete software-hardware platform is available. We focus on streaming applications, because they tend to be the most compute-intensive applications on mobile devices. Therefore, it is critical to optimize the hardware-software platform for streaming applications, as early as possible in the design process. As such, fast, accurate and early models are essential for hardware-software optimization.

Our design methodology is as follows. First, a measurement model is generated and executed, on the target processor, to predict the computation delays in an application. Next, the delays are annotated in the application code to generate a host-compiled model of the application. Our experiments show that such models can be generated and simulated at very high speed and accurately predict the computation load offered by the application. Our results with large streaming media applications, such as music and voice codecs, show that the estimation errors are less than 3.3%, while providing very high simulation speed. Therefore, using our models, embedded system designers can perform early optimizations to the system architecture with high confidence.

### ACKNOWLEDGEMENTS

I want to take this opportunity to thank and express my gratitude to my supervisor, Dr. Samar Abdi. His valuable guidance, helpful insight, and continuous support have given me a tremendous amount of experience and knowledge. Further, I thank him again for the constructive and positive discussions we had which helped me complete my research.

I am also thankful for all my professors for giving me an exceptional education. They have helped me build my background for conducting my research. I would also like to thank my examining committee members, Dr. Abdelwahab Hamou-Lhadj and Dr. Benjamin Fung, for reviewing my thesis and for giving me comments and suggestions.

Many thanks also go to my colleagues in the Embedded Systems Lab: Partha, Zaid, Ehsan, Karim, Ali, Paul, and Kazem for their suggestions, discussions, and support inside and outside the lab. Thank you very much for your patience and time when I get absorbed into debugging a program or when I ask for your suggestions on how to present my work.

Last but not least, I would like to thank my family and friends for their enormous support throughout my studies. I especially want to thank my parents and siblings for helping me get through this chapter in my life.

# CONTENTS

Lis	st of Ta	bles	viii
Lis	st of Lis	stings	ix
Lis	st of Fi	gures	x
Lis	st of A	crony	msxi
1	Intr	oduct	ion1
	1.1	Mod	lel Types1
	1.2	Met	hodology6
	1.2.	1	Model Inputs
	1.2.	2	Measurement Model Generator8
	1.2.	3	Logs Processing
	1.2.	4	Simulation Model Generator
	1.3	Rela	ited Work9
	1.3.	1	Traffic Generators9
	1.3.	2	Instruction-Set Simulator10
	1.3.	3	Transaction-Level Modeling11
	1.4	The	sis Contribution15
2	Rea	l-Tim	e Operating System Model16
	2.1	Mod	leling the Components17
	2.2	Task	Modeling
	2.2.	1	Scheduler Modeling19
	2.2.	2	CPU Time Consumption21
	2.3	Com	munication Modeling25
	2.4	Tim	er and Pulse Modeling28
3	Mo	deling	g Methodology
	3.1	Mea	surement Model Structure
	3.1.	1	Configuration File
	3.1.	1.1	Processor
	3.1.	1.2	RTOS

	3.1.3	1.2.1	Process	36
	3.1.1	1.2.2	Ports	37
	3.1.3	1.2.3	Interfaces	37
	3.1.1	1.2.4	Interrupt Handler	38
	3.1.3	1.2.5	Interrupt Controller	38
	3.1.3	1.3	Hardware Components	39
	3.1.3	1.4	Signals	40
	3.1.3	1.5	Connections	40
	3.1.3	1.6	Run Time	41
	3.1.1	1.7	Time Consumption	42
	3.2	Mea	surement Model Execution Semantics	42
	3.3	Logs	Processing	45
	3.3.3	1	Heuristics for Outliers	46
	3.3.2	2	Determining an Appropriate Block Delay	47
	3.4	Simu	lation Model Structure	48
	3.5	Simu	lation Model Semantics	50
4	Basi	ic-Bloo	k Level Annotation	52
	4.1	Appl	ication Timing at Basic-Block Level	52
	4.2	Anno	otation of Application Code	54
	4.3		surement Model Generation	
	4.4	-	Processing	
	4.5	Simu	lation Model Generation	59
5	Inte	_	el Level Annotation	-
	5.1	Anno	otation and Generation of Measurement Model	61
	5.2	Log F	Processing	62
	5.3	Simu	lation Model Generator	64
6	Expe	erime	ntal Results	67
	6.1	Use	Cases	67
	6.2	Mod	el Generation	70
	6.3		el Simulation	
	6.4		racy	
	6.4.3	1	Measurement Overhead for Basic-Block Annotation	76

e	5.4.2	CPU Load Estimation	77
6.5	De	esign Space Exploration	81
7 (	Conclus	sion and Future work	85
Appe	ndix		87
Α.	Conf	figuration File Schema	87
В.	Full A	Assembly Annotation of Figure 4.2	92
Refer	ences.		94

# List of Tables

Table 4.1 Blocks Identified for Figure 4.2	55
Table 6.1 Model Generation Time at the Basic-Block Level	70
Table 6.2 Model Generation Time at Inter-Kernel-Block Level	71
Table 6.3 Execution Speed of Application Models at the Basic-Block Level	72
Table 6.4 Execution Speed of Application Models at the Inter-Kernel-Block Level	73
Table 6.5 Estimation Accuracy of Simulation Models Using Average Induding Outliers	77
Table 6.6 Estimation Accuracy of Simulation Models Using Average Without Outliers	77
Table 6.7 Estimation Accuracy of Simulation Models Using Median Induding Outliers	78
Table 6.8 Estimation Accuracy of Simulation Models Using Median Without Outliers	78
Table 6.9 CPU Load Estimation Using Median delays and 10 bins in the PMF	80
Table 6.10 Average Absolute Errors for Observed Applications Tables 6.5 – 6.8	80
Table 6.11 Performance Increase from 20ms to 40ms Buffer Size	82
Table 6.12 Performance Increase from 40ms to 60ms Buffer Size	83
Table 6.13 Performance Increase from 60ms to 80ms Buffer Size	83
Table 6.14 Performance Increase from 80ms to 100ms Buffer Size	83

# List of Listings

Listing 2.1 Transition from Running to Ready	20
Listing 2.2 Transition from Running to Suspended	20
Listing 2.3: Transition from Suspended to Ready	21
Listing 2.4 Pseudo-Code for CPU Time Consumption	22
Listing 2.5 Pseudo-Code for Sending Message on Channel	26
Listing 2.6 Pseudo-Code for Receiving Message on Channel	27
Listing 2.7 RTOS Timers and Pulses	
Listing 2.8 Timer Emulation in SystemC	29
Listing 3.1: Configuration of Main Elements	33
Listing 3.2: Example of Processor's Elements	35
Listing 3.3: Hardware Components Example	
Listing 3.4: Signal Elements Example	40
Listing 3.5: Connection Elements Example	40
Listing 3.6: runtime Element Example	41
Listing 3.7: timeConsumption Element Example	42
Listing 4.1 START and STOP pseudo-code	55
Listing 4.2 START_RTOS_FUNC Pseudo-code	56
Listing 4.3 Simulation Model Subroutines Pseudo-code	60

# List of Figures

Figure 1.1 Impact of buffer sizing on computation load	3
Figure 1.2 Modeling Methodology	6
Figure 2.1 RTOS Modeling	16
Figure 2.2 Task States in the SystemC RTOS Model	19
Figure 2.3 Pre-Emption Modeling in Consume	24
Figure 2.4 Channel-Based Communication in RTOS	25
Figure 3.1 Measurement Model Generator	
Figure 3.2 Execution of Measurement Model on Target Processor	
Figure 3.3: Processor with process and RTOS	
Figure 3.4: Interface Example	
Figure 3.5: System View of Listing 3.5	41
Figure 3.6 Measurement of Application Execution Time on a Processor	43
Figure 3.7 Logged Data for a Block	45
Figure 3.8 Simulation Model Generator	48
Figure 3.9 Simulation Model on Host Processor	49
Figure 3.10 Block Delay Consumption in Simulation Model	50
Figure 4.1 Conditional Branching to Pseudo-Assembly Code	52
Figure 4.2 Annotated Pseudo-Assembly Code	54
Figure 4.3 Annotation to Omit Measurement of RTOS Model Functions	57
Figure 4.4 Overhead Cycles from Measurement	58
Figure 4.5 Distribution of Raw Measured Data of a Block using Basic-Block Annotation	59
Figure 5.1 Annotation of Measurement Code	61
Figure 5.2 Distribution of Raw Measured Data of a Block using Inter-Kernel-Block Annotation	on 63
Figure 5.3 Probability Mass Function	
Figure 5.4 Block Delay Annotation in Simulation Model	65
Figure 6.1 Tasks Sequence and Communication for MP3 + Vocoder	69
Figure 6.2 Manual Measurement of Overhead	76
Figure 6.3 Buffersize and CPU Load Trade-off	82

# List of Acronyms

HVP	High-level Virtual Platform
ISS	Instruction-Set Simulator
LLVM	Low-Level Virtual Machine
OS	Operating System
PMF	Probability Mass Function
QEMU	Quick EMUlator
QESC	QNX Emulation on SystemC
RESC	RTOS Emulation on SystemC
RTOS	Real-Time Operating System
TLM	Transaction-Level Models
WCET	Worst-Case Execution Time

# CHAPTER 1

# Introduction

One of the primary goals of embedded system design is to co-optimize the softwarehardware architecture for a given application. Designers typically tune various system parameters such as task priorities, scheduling policies, interrupt rates, types and numbers of hardware accelerators in order to optimize system performance for a given application. Model-based design advocates early system models that are executable and semantically represent the software-hardware architecture. The architectural parameters of the hardwaresoftware design are modified, and a new model is created to measure the impact of various design changes. A major challenge in modeling is the need for early and accurate performance estimation for a given software-hardware architecture. Designers are particularly concerned with determining the computation load offered by an application which is executed on the target processor. In this work, we have focused on streaming applications, since they tend to be the most compute intensive and power hungry applications on mobile platforms. Some examples of these applications are audio, video, and voice encoding/decoding.

## 1.1 Model Types

Various modeling methods have been proposed in the past to estimate software performance on a given hardware platform. Most common approaches include processor instruction set simulation, source-level analysis and worst-case execution time (WCET) analysis. Instruction-set simulation are extremely slow and, therefore, not well suited for extensive design space exploration. On the other hand, source-level analysis can be fast and accurate, but relies on the availability of the entire application's source code, which includes libraries. Moreover, details of the processor architecture must be available for timing analysis, which is not always practical for intellectual property (IP) confidentiality reasons. Lastly WCET analysis is excellent for hard real time systems, but provides pessimistic timing estimates, which may not be relevant for most embedded applications. Relying on WCET analysis can lead to overdesign.

In order to address the speed and accuracy issues of above techniques, we have proposed a measurement-based technique for modeling streaming applications on embedded platforms. Our models can be simulated to determine the load offered by the application to the target processor. Our technique capitalizes on the following two observations:

- (1) Typically, the processor core is available on an evaluation board or as a virtual prototype. The design challenge is to create an optimal platform by integrating on-chip buffers, hardware accelerators, and various interfaces with the processor together, and then porting the system software to the platform.
- (2) Most streaming embedded applications are designed as process networks, with little control flow inside the processes themselves. Consequently, the amount of computation inside a process depends largely on the amount of data being processed, as opposed to the data values themselves.

We present a motivating example of streaming application optimization. Figure 1.1(a) shows the message sequence chart of a typical multi-tasking streaming application executing on an embedded platform. The application is mapped to a target embedded processor, running a Real-Time Operating System (RTOS). The application is designed as a set of user tasks ( $T_1$ ,  $T_2$ , ...,  $T_n$ ) that communicate amongst themselves using local buffers (*buffer*<sub>1</sub>,



Figure 1.1 Impact of buffer sizing on computation load

*buffer*<sub>2</sub>,...*buffer*<sub>n</sub>). The input data stream is received from a file, network or other I/O, such as a microphone or camera, and stored in an on-chip hardware buffer (*IN buffer*). The output decoded/encoded data is stored in another on-chip hardware buffer (*OUT buffer*). The *OUT buffer* data is consumed by the network interface or other I/O, such as speakers or a display. An underflow interrupt from the *OUT buffer* is used to wake up the application tasks. The tasks then execute their respective encoding/decoding functions, write into their respective output buffers, and seek more data in their input buffers if needed.

An execution instance of the application in Figure 1.1(a) would proceed as follows. The underflow interrupt from the *OUT buffer* triggers the interrupt service routine (ISR), which copies any existing processed data from *buffer*<sub>1</sub> into *OUT buffer*. If there is insufficient data in *buffer*<sub>1</sub> to fill *OUT buffer*, the ISR sends a message to  $T_1$  to write more data into *buffer*<sub>1</sub>. Task  $T_1$  is therefore activated and reads from its input buffer (*buffer*<sub>2</sub>), processes it, and writes the output into *buffer*<sub>1</sub>. If there is insufficient data in *buffer*<sub>2</sub>,  $T_1$  sends a message to activate  $T_2$  to signal it to write more data into *buffer*<sub>2</sub> and so on. Finally,  $T_n$  is activated to read and process more raw data from *IN buffer*.

The sizing of the hardware buffers can have significant consequences on the performance of the system. For instance, if the *OUT buffer* is very small, it will take a short time to fill it, assuming a constant stream of incoming raw data in the *IN buffer*. As a result, the decoding/encoding delays will be small, leading to good quality of service. However, the *OUT buffer* data will also be consumed in a short time, leading to frequent underflow interrupts. Every underflow interrupt causes a context switch by the RTOS, and the cache needs time to be warmed up with tasks  $T_1$  to  $T_n$ . Once the *OUT buffer* is filled, the RTOS may switch to other user applications or kernel tasks, thereby potentially evicting  $T_1$  to  $T_n$  from the cache. When the next underflow interrupt arrives, the cache must again be warmed up with tasks  $T_1$  to  $T_n$ .

If we consider the reverse scenario where the *OUT buffer* is very large, it will take longer time to fill it, assuming a constant stream of raw data in the *IN buffer*. The decoding/encoding delays will be therefore larger, leading to a poorer quality of service. However, there will be less frequent underflow interrupts, more data processing done by the user tasks per underflow interrupt, and, fewer context switches. The cache will still need to be warmed up with tasks  $T_1$  to  $T_n$  after every underflow interrupt, but since the processor will be executing more iterations of the tasks per interrupt, the average cache behavior of the application will improve.

The consequences of buffer sizing are illustrated in Figure 1.1(b). The output buffer size is defined in terms of the time it takes to consume the data in the buffer. For instance, a buffer size of 20 ms corresponds to the amount of decoded data needed to playback 20 ms of music or voice. In the first trace, the *OUT buffer* size is set to 20 ms, leading to an average of t ms of task execution per underflow interrupt. Therefore, the computation load offered by the application is t/20. If the *OUT buffer* size is increased to 40 ms, the computation time per interrupt will increase to t' ms, since the amount of total computation per underflow interrupt will double. However, the improved overall cache performance, would imply that t' < 2t. Therefore, the overall computation load offered by the application will be less than t/20, resulting in lower power consumption.

The computation delay of the tasks, per underflow interrupt, may vary greatly with the state of the cache. As such, it is impractical to statically determine the computation delays. Therefore an executable simulation model is needed to predict these computation delays and to ascertain the average loads offered by the application. The system designer can use the predicted load to optimize the buffer size and other parameters for desired performance and quality of service.

This thesis presents a model methodology and its corresponding generation tools that are specifically designed for streaming applications. This model is aimed at facilitating early design decisions.



Figure 1.2 Modeling Methodology

## 1.2 Methodology

Figure 1.2 illustrates our modeling methodology. Our methodology requires two models, namely the measurement model and the simulation model. As shown above, these two models are generated by two independent model generators. The common inputs to both generators are: (i) the system configuration, which is an abstract representation of the target hardware platform in xml, (ii) the application source code, and (iii) a SystemC model of the target RTOS.

### 1.2.1 Model Inputs

The *System Configuration* input provides the information on how to model the targeted hardware platform. It also provides the necessary tasks, source files, and RTOS library that are needed by the *Application Software* and *RTOS Model* inputs. The application software is directed to the target RTOS and uses the RTOS API to manage the tasks and their communication. Furthermore, the RTOS model implements the scheduling policy of the target RTOS and the inter-task communication primitives in SystemC. Details on the RTOS model are further discussed in Chapter 2.

Consider the illustration of the application code in Figure 1.2. The application consists of tasks, each containing source code blocks that represent the task computation. During execution, the application task may execute from a block of code  $(b_1)$  to one of many source paths  $(P_{12}, P_{13}, P_{14})$  depending on the state of the application. The different paths may lead to different blocks of code  $(b_2, b_3, b_4)$ , which may take different amount of time to execute. Therefore, the application code needs to be annotated to identify which blocks are executed in order to accurately model the execution timing.

The model generators perform the annotation of the application code. Depending on the granularity of the annotation, the generators may produce different models of the annotated code. There are two types of annotation: fine-grained and coarse-grained. A fine-grained annotation would result in several invocations of the simulation kernel, leading to a slower simulation. A coarse-grained annotation will result in a faster simulation, but may lead to poorer accuracy. We consider fine-grained annotations at the basic-block level, and coarse-grained annotations at computation blocks between RTOS kernel calls.

A basic-block is, by definition, a sequence of instructions such that if the first instruction is executed, then all the instructions in the basic block are executed. They have a single point of entry and a single point of exit during execution. The basic-blocks are not visible at the source level of the applications and can only be observed at the assembly level. An interkernel-block is defined as a sequence of application code between two RTOS kernel calls. During an execution of an inter-kernel block, no RTOS kernel call is made. So, we consider the inter-kernel-blocks to be atomic for annotation purposes. An inter-kernel-block typically consists of multiple basic-blocks. Hence, inter-kernel-blocks are coarser in granularity. Annotating at the inter-kernel-block level is acceptable for our methodology because (i) streaming applications have limited control flow and (ii) RTOS calls are easy to identify while parsing the source code. The method for annotation at the basic-block and inter-kernelblock levels is further discussed in Chapter 4 and 5 respectively.

#### 1.2.2 Measurement Model Generator

The *Measurement Model Generator* produces a SystemC model of the application, with annotated hooks to measure and log the execution times of each block. The application model is linked with the RTOS model to produce a binary that can be executed on a base operating system (OS), such as Linux, running on the target processor. After execution, a log of block delays, over several iterations, is obtained.

#### **1.2.3 Logs Processing**

The computation delay for a single block may vary across iterations due to a variety of environmental factors, such as processor state, cache state, or scheduling in the base OS. Hence, the delays for each block are processed to account for such variations. Each block's logged data is filtered to take the above variations into account. The logged delays for each block are extracted as a single average or median delay, or as sets of stochastic delays. The processing of the logs is discussed in Chapter 3, Section 3.3.

#### 1.2.4 Simulation Model Generator

The block delays from the processed logs are annotated to the application code. The annotated application is linked with the SystemC model of the RTOS and hardware platform to generate the simulation model. The simulation model is then executed on a host machine to obtain estimated CPU load offered by the application. The CPU load estimation is performed

on the host machine to accelerate the simulation speed, since it is faster than the target machine.

### **1.3 Related Work**

There are three fundamental modeling technologies that can be used to estimate resource consumption by embedded software before hardware availability: (i) traffic generators, (ii) instruction-level simulation models, and (iii) host-compiled timed transaction-level models (TLMs).

#### 1.3.1 Traffic Generators

Traffic generators are used very early in the design flow, even before the application source code is available. The goal is to exercise the underlying hardware, or its model, with stochastic execution scenarios that can be expected from the application. Traffic generators typically inject data packets or messages into the hardware model to replicate the data communication traffic generated by the software application. The generated traffic is based on estimation of the application's workload. Naturally, traffic generators are not an accurate representation of embedded software execution and are of limited use. However, they can be useful.

The methodology in [1] proposes to three approaches to study the system performance before the hardware design is complete. The first approach models the software behavior with some level of the hardware descriptions. The second uses a trace to measure and replicate the transactions between the processor and the external memory. The third approach blends the first two methods as a stochastic traffic generator to estimate the pre-silicon system performance. The three approaches generate more accurate results only as the description of the software and hardware architectural designs become more available.

#### 1.3.2 Instruction-Set Simulator

Instruction-level simulation models are often used as virtual hardware platforms for software development, before the hardware is available. The instruction-set simulator is usually programmed in a high-level language to model the processor. The processor model typically emulates instruction fetch, decode, execution, and memory operations. The execution of the instructions transforms the modeled processor's registers, thereby providing visibility into the processor state. The accuracy of an instruction set simulator depends on the abstraction level at which the processor has been modeled. The instruction set-simulator is typically slow as it has to interpret every instruction in the application binary at the cycle accurate level [2-3].

The research presented in [2, 14, 28] uses analytical and simulation-based approaches to obtain accurate and fast performance estimations. The analytical approach considers the worst case/best case execution time of basic-blocks. The simulation-based approach executes the input application code on the target architecture, modeled in SystemC, and back-annotates the basic-block delays from the analysis. The back-annotation also considers the timing delays for branch prediction and cache hits and misses. These approaches' case studies show a speedup of up to 91% from the ISS approach and an error rate of 4%-7% which is in the same range of an ISS.

The work in [3, 29] uses a High-level Virtual Platform (HVP) to provide early software development. HVP models execution of applications on multiprocessor System-on-Chips. Moreover, HVP mainly focuses on supporting software development, meaning the

10

application code can be modified in early design stages. The HVP framework models the application code as tasks and the processors as virtual processing elements. Similar to our methodology, an abstract OS model must be provided to schedule the tasks to execute on the virtual processing elements. The tasks can be provided as native C code or ISS tasks. The latter requires an ISS to be available. It is assumed that an ISS can be used when a processor has already been determined in early design stages. The experimental results were able to predict the application execution speed up by using multiple virtual platform elements to determine the trade-off between the execution speed and the number of processors. The experimental results also showed that using the native C code provided a much faster simulation than the instruction-set simulators.

While both techniques in above provide accurate and fast performance estimations, they need the virtual platforms to develop the system software, such as RTOS and drivers. In order to use the instruction-level virtual platform for estimating embedded software performance, the RTOS must already be available on the virtual platform. In contrast, our modeling methodology assumes that the target processor is already available, and that a new hardware platform is to be designed around the processor. This is a more realistic assumption, since generations of products do not typically change processor types. The hardware platform transitions in most industrial design methodologies are more gradual and incremental. Furthermore, our modeling methodology is several orders of magnitude faster than ISS-based models.

#### **1.3.3** Transaction-Level Modeling

Transaction-level modeling is high-level approach where the communication and the implementation of platform architecture are separated. TLMs are typically developed in

system-level design languages, such as SystemC, and can be executed on a host machine [4]. There have been several approaches to automatically generate TLMs from a high level description of the target hardware, but a timed RTOS model is needed to accurately estimate the application execution time on a given platform [5-7].

The work in [5] presents the automation of TLMs for multicore platforms. Their approach generates two TMLs: a functional model and a timed model. The functional model ensures the application semantics are still preserved after distributing the workload onto multiple cores. The timed model provides the performance of the application platform design. They have used the low-level virtual machine (LLVM) to compile the basic blocks, annotate the timing delays code, and disassemble the annotated blocks. The disassembled annotated block can be recompiled on the host machine to speed up the simulation time. The processing elements uses cycle delays to consider the execution of the pipelined instructions, ALU computation, cache hit/misses, and branches. Their experimental results proved that using the TML simulations were much faster than register-transfer level (RTL) and instruction-set simulations and that the number of executed cycles were accurate (error of 4.5%-7%). However, the number of executed cycles is not enough to predict the application performance due to the multicore parallelism. Hence, a timed RTOS model was incorporated into the timed model to prove that scheduling the tasks on multiple cores was needed to provide accurate timing.

The approach presented in [6] integrates the Quick EMUlator (QEMU) [23] and SystemC [24] to create a TLM for multiprocessor system on chip. The QEMU component is used to simulate the processor's behavior (fetch, decode, execute, branch, and cache-hit/miss) for both the native C code and the low-level instructions. The native C code can be first cross-

compiled for target, then annotated, and finally dissembled back for the host like [5]. SystemC is used to model the other platform components such as hardware accelerators and to manage the QEMU component as a module. The management allows execution concurrency with respect to the SystemC hardware models. The approach has deducted that combining QEMU and SystemC can provide an application's performance with accuracy similar to that of an ISS, with the exception that the simulation will be faster. However, the timing model of native C tasks which are mapped to the same processing element is not accurate because their execution is not interleaved on the processing element.

The methodology in [7] proposes to use an actor-oriented model to describe the functional behavior of an application. This approach uses SysteMoC (a SystemC based library for actor-oriented models), which allows the actors to communicate through channels. The actors are mapped into processing elements which uses ISSs in order to simulate the computation. Although no RTOS was cited, it was mentioned that a scheduler implementation is required when a processing element has multiple actors.

The above mentioned papers provide an excellent base for embedded application performance estimation. The common complication for the above approaches is the modeling of concurrency of multiple tasks/actors on a single processing element. Since their methodologies use SystemC, a model of the RTOS scheduler can also be developed similarly to [8–11] to manage task concurrency. However, the above methods require the availability of a detailed processor data model for timing accuracy. Such data may not be readily available due to intellectual property protection of the processor.

The RTOS models presented in [8-11] had the common goal to model the dynamic scheduling of the application tasks. Task preemption, synchronization, interruption, and

execution were implemented using system-level design languages such as SystemC [24], SpecC [25], and Metropolis [26]. We are particularly interested in the task modeling to see how a processing element can be scheduled to model the execution of a single task at a time. Moreover, we want to determine the execution time each task takes on the target processor.

RTOS models can also incorporate timing delays of kernel calls as presented in [12–13]. The incorporated timings delays are dependent of the design of the processing element, which may not be available in early stages. The number of cycles required by the processing element can be computed based on the data flow execution of the instructions. It may be possible to manually insert the delay annotations in the source, but such approach is not practical for large applications. Hence, an automated solution is desirable to quickly obtain the accurate timing delays. In host-compiled TLMs, timing is added to the application source code, and the annotated application is linked to an RTOS model for simulation [14–17]. Typically, the timing is annotated at the function or basic-block level [18]. Source-level simulation techniques can be used for accurate instruction and data cache simulation [19-20]. However timing annotation in TLMs requires a data model of the processor, which is not always available due to intellectual property concerns. Moreover, the entire source of the application, including libraries, must be available. As such, source-level timing annotation, based on static code analysis, is not always practical. The contribution presented in this thesis builds upon previous work on RTOS modeling in SystemC. We use an executable measurement model to determine timing of application code. Therefore, our model can use the RTOS targeted application code as is without requiring the library sources. We do not model the timing of the RTOS primitives themselves, since the RTOS overhead is observed to be in the range of 2-5% for streaming applications [27].

14

### **1.4** Thesis Contribution

This thesis presents a novel methodology and supporting tools which provides early and accurate modeling of streaming embedded applications. The main contributions this paper presents are:

- A novel modeling methodology for embedded systems, based on measurement on the target platform and simulation on a host system. The methodology enables early and reliable design space exploration.
- Design and implementation of tools for automatic generation of embedded system models from abstract hardware and system software specification. The model automation replaces the time-consuming and error-prone process of manual modeling of streaming applications on embedded platforms.
- Implementation of automatic timing annotation in embedded system models at different levels of granularity, and evaluation of trade-offs between model execution speed and accuracy for different annotation granularities. The strengths and weaknesses of fine-grained versus coarse-grained annotations are analyzed.
- Demonstration of early, fast, and reliable design space exploration with automatically generated models of industrial size streaming applications, such as image compression, MP3 playback, and voice codecs.

# **CHAPTER 2**

# **Real-Time Operating System Model**

This chapter describes the implementation of the RTOS model needed for early performance estimation. The chapter consists of four sections. The first section describes the modeling of the system components of the target platform in SystemC. The second section describes the modeling of applications tasks from perspective of the RTOS scheduler and the time consumption on target CPU. The third section describes the modeling of software timers in the RTOS model. Timers are a common feature of real-time embedded software, so their modeling is highly pertinent to our work.



Embedded Software Execution Environment

System Level Model

Figure 2.1 RTOS Modeling

## 2.1 Modeling the Components

Figure 2.1 illustrates the modeling of the target RTOS in SystemC. We start with the application software code and platform specification, shown on the LHS, and derive a functionally equivalent SystemC model, as shown on the RHS. The application software is targeted to an RTOS, running on a target embedded processor. Concurrent application tasks A, B, and C, are captured as threads in the user code. Inter-task communication is implemented using the message passing API of the RTOS. The target processor is typically part of a larger hardware platform consisting of other system components such as processors, DSPs, custom hardware accelerators, memories and I/O. It is assumed that executable C/SystemC models of other system components are available.

On the RHS, we have a derived model of the system, which can be compiled and executed on the host (typically a PC running Linux) before the hardware is available. The target processor is abstracted as a SystemC module (shown as a rounded box) that encloses the application model and the RTOS model. Here, A', B', and C' are functionally equivalent abstractions of A, B, and C, and are implemented as SystemC tasks. Since SystemC does not natively support any RTOS primitives for scheduling and communication, a RTOS Emulation on SystemC (*RESC*) layer is modeled on top of the SystemC kernel. RESC supports prioritybased scheduling and inter-task communication, similar to a typical RTOS.

We chose SystemC as the modeling language since it provides the necessary constructs, namely concurrency, event-based synchronization, timing and object-orientation, needed to create an executable system level model [4]. For example, a SystemC module (SC\_MOUDULE) can model a hardware component, which can have its internal logic and external ports to connect to other modules. A SC\_MODULE can also create the

17

representation of threads (SC\_THREAD). Each thread is executed one at a time until it has to wait on for a specific amount of SystemC time (sc\_time) or event (sc\_event). Once the thread has finished waiting, it continues to execute until another wait has been called or until it terminates. SystemC simulates the execution of the threads to run concurrently, but the threads are seen to execute in parallel with respect to SystemC's time.

SystemC natively supports timing and events, therefore timers and pulses in the application can be abstracted using native SystemC constructs. This part of the model has a direct dependency on the SystemC libraries. Finally, we do not explicitly model the memory management of the target platform or the I/O needed for debugging. These services are used from the run-time system available on the host.

### 2.2 Task Modeling

The application tasks are dynamically created in the RTOS-targeted software using pthread\_create. In SystemC, we model all tasks as SC\_THREADs, which can be dynamically created using sc\_spawn method of the SystemC kernel. SystemC, however, does not have any provisions for priority based scheduling, since it is simply a discrete event simulation engine. Therefore, we define a new class *RESC\_task*, which stores the task ID, priority, state, an activation event, and a handle to the corresponding SystemC thread for each task. Since the RESC\_task structure contains a SystemC event, it cannot be instantiated dynamically to reflect dynamic task creation in a RTOS. We bypass this problem by creating a static pool of tasks. RESC implements functions for dynamically allocating tasks from this pool and managing them, similar to creation and management of the corresponding RTOS tasks. As opposed to user tasks that are spawned by the application, the system also consists of interrupt service routines (ISRs) that are triggered by hardware interrupts. The interrupts themselves

are modeled as SystemC signals, generated by the SystemC hardware modules. The ISRs are modeled in SystemC as SC\_METHODs that are sensitive to the respective hardware interrupt signal.

#### 2.2.1 Scheduler Modeling



Figure 2.2 Task States in the SystemC RTOS Model

The RESC scheduler manages the states of the application tasks as shown in Figure 2.2. We define four states that a task may be in during execution: *RUNNING*, *READY*, *SUSPENDED*, and *TERMINATED*. Upon creation, a task is in READY state. Whenever the scheduler is executed, the highest priority ready task is moved to the RUNNING state. Conversely, a running task may be pre-empted by a higher priority task that becomes ready due to a scheduling event such as an interrupt, timer pulse or resource availability. Running tasks may become suspended waiting for resource or a timer pulse. Finally, a running task may be terminated upon exit. RESC defines private methods for the transitions shown in Figure 2.2. These methods are used by the user-level API of RESC to manage task states.

void RESC::Running2Ready ()

- 1: *RESC\_task* \*t = ACTIVE;
- 2:  $t \rightarrow State = READY;$
- 3: ACTIVE = GetHighestPriorityReady();
- 4: *if* (ACTIVE != t) {
- 5: ACTIVE  $\rightarrow$  Activation.*notify*();
- 6: *wait* (t $\rightarrow$ Activation);
- 7: } // end if

Listing 2.1 Transition from Running to Ready

Listing 2.1 shows the pseudo-code for transition from running to ready state. The currently running task (pointed to by *ACTIVE*) that calls this method is stored in task pointer t, and its state is changed to READY. The ACTIVE task pointer is updated to point to the highest priority ready task. The method *GetHighestPriorityReady* sets the state of the new active task to RUNNING.

If the new active task is different from the caller, its execution in SystemC is enabled by notifying its Activation event. The caller task is blocked on its activation event, indicating it is ready to run as soon as it becomes the highest priority task. If the caller remains the highest priority ready task, it does not need to wait on its activation event and may proceed in the RUNNING state.

void RESC::Running2Suspended ()
1: ACTIVE $\rightarrow$ State = SUSPENDED;
2: ACTIVE = GetHighestPriorityReady();
3: $if(ACTIVE != NULL)$ {
4: ACTIVE $\rightarrow$ Activation. <i>notify</i> ();
5: } // end if
Listing 2.2 Transition from Running to Suspended

Listing 2.2 shows the pseudo-code for the transition from running to suspended state. The state of the caller task, which is pointed to by ACTIVE, is set to SUSPENDED. The state of the highest priority ready task, if one exists, is set to RUNNING and its Activation event is notified. Note that the caller task is not blocked in SystemC by this method, because it will be blocked on a resource or timer pulse once it returns from this method. This scenario will be seen in Sections 2.3 and 2.4.

void RESC:: Suspended2Ready ()
1: RESC_task *t = GetTask (sc_process_handle());
2: $t \rightarrow \text{State} = \text{READY};$
3: <i>if</i> (ACTIVE == NULL)
4: ACTIVE = GetHighestPriorityReady(); // run caller
5: else
6: <i>wait</i> (t $\rightarrow$ Activation);
Listing 2.3: Transition from Suspended to Ready

When a suspended task resumes execution in SystemC, it calls RESC's Suspended2Ready method, shown in Listing 2.3, to restore its state back to READY. The pointer to the caller's RESC\_task object is obtained using the current SystemC process handle as an index. The state of the caller task is set to READY. If all the tasks were suspended before this task was woken up, the ACTIVE pointer would be NULL, indicating that the CPU is idle. If so, the state of the caller task is set to RUNNING and it continues execution in SystemC. Otherwise, the task waits for its Activation event like any other ready task.

#### 2.2.2 CPU Time Consumption

The system level model must contain delay annotations in the tasks to model their CPU time consumption. Modeling the time consumption of a given code segment on a hardware

platform is an inherently difficult problem. The problem has been actively researched and there are some well-known methods for predicting software execution time based on a model of the hardware. Typically, a prototype board with the processor core is available. Therefore, in order to obtain accurate delays, we measure the execution time of the computation blocks between the kernel calls on the processor. The delays are then back annotated to the model.

SystemC supports incorporating delays into the model by using wait statements. However, this construct is insufficient for modeling the CPU time consumption of concurrent tasks executing on an RTOS. The wait statements in concurrent tasks consume the delays concurrently, as opposed to interleaving of concurrent tasks on a processor. We solve this problem by defining a *Consume* function in RESC. The Consume function is used by the application tasks to consume time on the CPU.

void RES	C::Consume ( <i>sc_time</i> t)
1: sc	<i>time</i> TimeRemaining = t;
2: <i>sc</i>	<i>_time</i> Start, Delta;
3: wi	hile (true){
4:	<pre>Start = sc_time_stamp();</pre>
5:	wait (TimeRemaining, ScheduleEvent);
6:	Delta = <i>sc_time_stamp()</i> - Start;
7:	TotalBusyTime += Delta;
8:	<i>if</i> (Delta == TimeRemaining) break;
9:	TimeRemaining -= Delta;
10:	Running2Ready();
11:}	// end while
12: Ru	unning2Ready();
<b>.</b>	

Listing 2.4 Pseudo-Code for CPU Time Consumption

Listing 2.4 presents the pseudo-code of the implementation of the Consume function. The variable *TimeRemaining* keeps track of the amount of time that remains to be consumed if the caller task is pre-empted during the consume call. Another RESC variable, *TotalBusyTime*, keeps track of the total time consumed on the CPU during system execution. This metric is useful for estimating system-level power and for performing optimizations. A SystemC event, named *ScheduleEvent*, is defined in RESC to notify rescheduling of tasks due to interrupts or timer pulses.

Variable *TimeRemaining* is initialized to t, the amount requested by the caller task (line 1). The current timestamp is stored and the SystemC wait on *TimeRemaining* and *ScheduleEvent* is called (lines 6-7). If the SystemC logical time has advanced by *TimeRemaining*, or *ScheduleEvent* is notified before that, the wait call returns. The new SystemC timestamp is used to determine the amount of time consumed (Delta). The consumed time is added to the *TotalBusyTime*. If the *ScheduleEvent* is notified before the remaining time is consumed, the variable *TimeRemaining* is decremented by the consumed time and the task state is updated to READY. The concurrent wait on time and event avoids any timing errors due to the granularity of time delays annotated in the task. We iterate until all requested time has been consumed.

It is possible that *ScheduleEvent* is notified at the same logical time as the end of time consumption. To take care of this scenario, we call Running2Ready before exiting the consume method. This will ensure that if *ScheduleEvent* was indeed notified, the new highest priority task will be executed. Otherwise, no other task would have become ready during execution of the wait, and the current task will remain in the RUNNING state.

23



Figure 2.3 Pre-Emption Modeling in Consume

Figure 2.3 illustrates how the RESC Consume function handles a task pre-emption scenario. We consider an application with two tasks T1 and T2, with T1 having a higher priority than T2. Further, we assume that, initially, task T1 is suspended, waiting for a resource to become available, while T2 is running. T2 makes a consume call for 20 time units to model its CPU time consumption. After 10 time units, an external interrupt signal triggers the execution of an ISR. The ISR notifies ScheduleEvent, which pre-empts T2 by interrupting the wait statement in T2's consume call (Listing 2.4, line 5). The remaining time for T2's consume call is calculated (TimeRemaining = 10 units) and T2 is moved to the READY state (Listing 2.4, lines 9-10).

The ISR executes with the highest priority and wakes up T1, thereby moving T1 to the READY state. Once the ISR exits, the scheduler is run to determine the highest priority READY task. Since T1 is the highest priority READY task at this time, its *Activation* event is notified by the ISR's exit method. T1 runs and consumes 5 units and is suspended on another

resource by calling Running2Suspended. That leaves T2 as the only READY task. T2's *Activation* event is notified by T1's Running2Suspended call (Listing 2.2, line 4). Therefore T2 resumes its consume call and enters the second iteration (Listing 2.4, Line 3), where it consumes time for the remaining 10 units (Listing 2.4, Line 3) and exits. Task interleaving and pre-emption is therefore modeled quite simply and accurately.

### 2.3 Communication Modeling

User tasks may communicate with each other using several communication semantics. The basic communication mechanism that is used to support these semantics is message passing. The message passing between tasks is supported by channels that buffer the communicated data. In this section, we describe implementation of message passing channels in RESC.



Figure 2.4 Channel-Based Communication in RTOS

Figure 2.4 illustrates the double-handshake semantics of channel communication in a RTOS. As seen in the figure, the receiver creates the channel and opens it to receive data. Concurrently, the sender also opens the channel, and calls the *MsgSend* method to copy the sender's data into the send buffer. After writing, the sender waits for a reply notification from
the receiver. At the other end, the receiver calls *MsgReceive*, which blocks until the sender has written to the send buffer. After returning from MsgReceive, the receiver calls *MsgRead* to copy data from the send buffer into its own memory. The read may be done from any offset and can be of any size. Finally, the receiver calls *MsgReply* that writes into the reply buffer of the channel and notifies the sender. The sender task is unblocked, reads the reply buffer and returns from *MsgSend*.

RESC implements the message passing communication of an RTOS using a *RESC\_channel* class defined on top of SystemC. The double handshake synchronization is implemented using a boolean flag (*SendFlag*) and events, (*SendEvent* and *ReplyEvent*) in RESC\_channel. Similar to RESC\_task, we cannot dynamically instantiate objects of type RESC\_channel due to restrictions on dynamic creation of SystemC events. We bypass this problem by creating a static pool of channels. The channels are allocated from this pool whenever a channel is created in the user code.

void RESC:	MsgSend	( <i>int</i> channel_id,	)
------------	---------	--------------------------	---

- 1: Copy data into send buffer
  - 2: CHANNELS[channel\_id].SendFlag = true;
  - 3: CHANNELS[channel\_id].SendEvent.notify();
  - 4: Running2Suspended();
  - 5: *wait* CHANNELS[channel\_id]→ReplyEvent;
  - 6: Suspended2Ready();
- 7: Copy data from reply buffer;

Listing 2.5 Pseudo-Code for Sending Message on Channel

Listing 2.5 illustrates the MsgSend implementation in RESC. The array CHANNELS refers to the pool of channels, indexed by variable *channel\_id*. MsgSend copies the data pointed to by the sender into the send buffer. It then synchronizes with the receiver by setting

SendFlag to true and notifying SendEvent to indicate that the receiver can now read from the send buffer. The sender waits for the reply by moving itself to the suspended state and waiting on ReplyEvent. Once the receiver has written to the reply buffer and notified ReplyEvent, the sender returns to the ready state and eventually copies over data from the reply buffer.

void RESC:: MsgReceive (int channel_id,)
1: if (!CHANNELS[channel_id].SendFlag) {
2: Running2Suspended();
3: wait (CHANNELS[channel_id].SendEvent);
4: Suspended2Ready();
5: }
6: CHANNELS[channel_id].SendFlag = false;
Listing 2.6 Pseudo-Code for Receiving Message on Channel

Listing 2.6 illustrates the MsgReceive implementation in RESC. If SendFlag is true, the receiver knows that the send buffer has already been written. Therefore, it simply resets SendFlag and proceeds to read the data. Otherwise, the receiver waits on SendEvent until the send buffer is written. The receiver puts itself in the suspended state before the wait and returns to the ready state after the wait.

Channel based communication is reliable and encouraged for real-time applications. A common design issue is the sizing of buffers in the channels. Very large buffers may cause memory bottlenecks, while very small buffers may require several transactions, thereby increasing the time overhead caused by handshake synchronization. System-level models, such as the one described here, provide performance feedback that can help make design decisions on buffer sizing.

## 2.4 Timer and Pulse Modeling

1: *sigevent* pulse;

2: *timer t* timer;

3: *itimerspec* tspec;

- 4: SIGEV\_PULSE\_INIT (&pulse, channel,...);
- 5: tspec.it\_value.tv\_sec = 0;
- 6: tspec.it value.tv nsec = 10 \* 1e6; //10 ms
- 7: tspec.it\_interval.tv\_sec = 0;
- 8: tspec.it interval.tv nsec = 20 \* 166; //20 ms
- 9: timer settime(timerid, 0, &tspec, NULL);

## Listing 2.7 RTOS Timers and Pulses

In an RTOS, a timer can be set up to periodically send pulses to a user task at given time intervals. Listing 2.7 illustrates the key aspects of timer creation and setup in an RTOS. Timers use system events (*sigevent*) as pulses that are sent over a channel (lines 1-4). A timer specification (*itimerspec*) consists of an initial wait time (*it\_value*) for the first pulse and an interval wait time (*it\_interval*) for subsequent periodic pulses (lines 5-9). The timer initialized in Listing 2.7 sends pulses over channel at times 10ms, 30ms, 50ms, 70ms and so on, after *timer settime* is called.

In order to emulate a timer in the SystemC model, we define a *RESC\_timer* class and a corresponding SystemC thread (distinct from application tasks), whose functionality is shown in the timer method in Listing 2.8. Similar to tasks and channels, we define a pool of RESC\_timer objects. A timer pulse is modeled as an event in the RESC\_timer class. Corresponding to a timer creation in the RTOS, the SystemC model allocates a timer object from the timer pool. The timer\_settime method corresponds to dynamic creation of the timer thread using *sc spawn*.

1:	<pre>void timer(timer_id, it_value, it_interval){</pre>
2:	wait(it_value);//initial wait values
3:	while(true){
4:	Timers[timer_id] $\rightarrow$ pulse. <i>notify</i> ();
5:	ScheduleEvent.notify(0);
6:	wait(it_interval);//periodic wait values
7:	}// end while
8:	}
9:	<pre>void RESC::Wait4TimerPulse (int timer_id){</pre>
10	Running2Suspended();
11:	$wait$ (Timers[timer_id] $\rightarrow$ pulse);
12:	Suspended2Ready();
13.	; }
	2: 3: 4: 5: 6: 7: 8: 9: 10: 11:

Listing 2.8 Timer Emulation in SystemC

The timer thread operation is fairly straightforward as seen in Listing 2.8. The timer waits for the initial wait time as defined in the timer specification ( $it\_value$ ), followed by an infinite while loop (lines 2-3). Inside the loop, the pulse event for the specific timer is notified to wake up the task sensitive to the timer pulse. This is followed by a delta cycle delayed notification of the scheduling event. The timer thread then waits for the interval period ( $it\_interval$ ) until the next periodic pulse. The task sensitive to the timer calls the Wait4TimerPulse method defined in RESC, as shown in Listing 2.8 (lines 9-13). The waiting task is suspended while waiting for the timer pulse event. The ScheduleEvent notification (line 5) in the timer thread is delta cycle delayed to allow the waiting task to update its state to READY before the active task is moved to the ready state (Listing 2.4, line 10) and forces a rescheduling of tasks.

# CHAPTER 3

## **Modeling Methodology**

In this chapter, we discuss the details of our modeling methodology, including the model generation tools, log processing, and CPU load estimation. First, we present the steps of model generation from the input specification of the embedded hardware and software, as well as the application code and RTOS model. Next, the semantics of the generated models are explained in further detail. The measurement model's execution produces a log of measured block delays. The generated logs are then processed, and input to the simulation model generator. Finally, the simulation model uses the processed log to annotate timing into the application code and generate a SystemC model for simulation on host.

## 3.1 Measurement Model Structure



Figure 3.1 Measurement Model Generator

The *Measurement Model Generator*, shown in Figure 3.1, takes three inputs, the *System Configuration*, the *Application Software*, and the *RTOS Model*, to produce the executable measurement model. The *System Configuration* file specifies the application software source files, the RTOS model, and the target hardware platform. The configuration schema, using

W3's XML Schema format, is defined in Appendix A. The generator performs the following steps:

- (1) Parse the *System Configuration* file and create an internal data structure to store the information needed to generate the Measurement Model.
- (2) Generate the SystemC models of the hardware platform, including the processor and peripherals.
- (3) Instantiate the RTOS model in the processor.
- (4) Instantiate the tasks of the applications in SystemC modules corresponding to the processor. The body of each task is filled with calls to the corresponding task entry functions in the application code.
- (5) Modify each file of the application source.
  - a. Parse and identify target OS function calls which have a different implementation than the base OS. The identified target RTOS function calls are replaced by the functions in the RTOS model to disambiguate the functions of the base OS and the modeled target OS.
  - b. Do a second parse to annotate measurement code at the specified granularity:
    - i. Identify blocks (basic-block or inter-kernel level).
    - ii. Insert a *START* subroutine/function at the beginning of a block and a *STOP* subroutine/function at the end of the block. The *START* begins measuring and identifying a block. The *STOP* stops and records the measurement for the identified block.
- (6) Generate the Makefile for the compilation and execution of the output model.

Figure 3.2 shows how measurement model can be generated. This measurement model is executed on the target machine to obtain accurate measurements. The annotated application tasks, A, B, and C, modeled as SystemC threads, are executed using the RTOS model. The function f1(), in application task A, is shown with a disambiguation box to illustrate whether



Figure 3.2 Execution of Measurement Model on Target Processor

or not an implementation of *f1* exists in the RTOS model. Step (5)a above would have modified the function name if it is one of the modeled RTOS functions. The measurement annotation uses the hardware timer to obtain the timing delays of the executed code. The *RTOS* model and the *Models of Other Processors/Peripherals* are then executed on top of the SystemC kernel which is run by using the base OS.

#### 3.1.1 Configuration File

The model generators require the hardware-software specification, as defined in the system configuration file. The configuration file defines the application source code, the simulation environment and the hardware platform set up. It begins by defining the SystemC path (*systemcPath*), the compiler (*compiler*), the system layout (*system*), and the executable name (*executable*). With the exception of *system*, the defined elements are simple string formats which are used when generating the *Makefile*. The *system* element contains the main

components at the system layout level which will be explained shortly. The following listing illustrates an example on the use of the other three elements:

```
<configuration>
       <systemcPath>/usr/systemc</systemcPath>
       <compiler>g++</compiler>
       <system>
              <processor>
              </processor>
              <hardware>
              </hardware>
              <signa⊳
              </signa⊳
              <connection>
              </connection>
              <runtime>
                     . . .
              </runtime>
              <timeConsumption>
              </timeConsumption>
       </system>
       <executable>application.exe</executable>
</configuration>
```

Listing 3.1: Configuration of Main Elements

In Listing 3.1, the element *systemcPath* defines the directory path of SystemC to be in */usr/systemc*. Since the configuration is for SystemC, the compiler has to be specified to  $g^{++}$  in order to compile the source files with generated SystemC system files. The *executable* element specifies the executable filename of the measurement model, while the sub-elements of *system* are not listed because they are too complex to illustrate as a single listing. These sub-elements are broken down to simplify the generation of the model.

#### 3.1.1.1 Processor

The first sub-element *processor* specifies the RTOS model and the software application's process. The user should provide the processor *type* and *name*, which will be assigned to the variable type and the variable name, respectively, for the component in the system specification. The *type* will also be used to create a header file which defines a SystemC module. The processor can also specify various parameters such as the interrupt handler, controller, ports, and hardware interfaces. The following listing shows an example with the sub-elements of *processor*.

```
<processor>
   <type>processor</type>
   <name>p</name>
   <frequency>500e6</frequency>
   <RTOS>
          <type>QNX</type>
          <model>OESC</model>
          <name>qesc</name>
          library>../../../QESC/gesc.o</library>
          libHome>../../../QESC</libHome>
          <policy>SCHED FIFO</policy>
   </RTOS>
   <process>
          <name>startingThread</name>
          <header>aHeaderFile.h</header>
          <cfile>aCFile.c</cfile>
          <mainFunction>renamed main</mainFunction>
          <task>
                 <taskEntry>t1</taskEntry>
                 <priority>10</priority>
          </task>
          <task>
                 <taskEntry>t2</taskEntry>
                 <priority>20</priority>
          </task>
   </process>
   <port>
          <name>portA</name>
          <type>int</type>
```

```
<direction>out</direction>
   </port>
   <port>
          <name>interfaceIF</name>
          <type>Interface IF</type>
          <direction>port</direction>
   </port>
   <interface>
          <header>Interface IF.h</header>
          <function>
                  <fname>F1</fname>
                  <return>void</return>
                  <IF portName>interfaceIF</IF portName>
                  <parameterType>void</parameterType>
          </function>
   </interface>
   <interruptHandler>
          <name>IH</name>
          <header>isr.h</header>
          <case>
                  <sourcePort>interrupt1</sourcePort>
                  <subroutine>isr</subroutine>
          </case>
   </interruptHandler>
   <interruptController>
          <type>IC</type>
          <name>ic</name>
          <interruptPort>
                  <name>interrupt1</name>
                  <type>book/type>
                  <direction>in</direction>
          </interruptPort>
   </interruptController>
</processor>
```

Listing 3.2: Example of Processor's Elements



Figure 3.3: Processor with process and RTOS

### 3.1.1.2 RTOS

As Listing 3.2 shows, the *RTOS* element the RTOS type (*type*), its model (*model*), including its library name (*library*) and path (*libHome*), its variable name (*name*), and its scheduling policy (*policy*). The library name and path of the RTOS model needs to be defined with respect to the directory in which the codes are being generated. Figure 3.3 shows how the RTOS model is linked with the processor.

#### 3.1.1.2.1 Process

The next element of the processor is the *process*. A processor needs at least one process to perform some computation. The *process name*, corresponds to the SC\_THREAD that will be created for the process inside the processor SystemC module. The SC\_THREAD initializes a task which is specified by the element *mainFunction*, through the RTOS model. The *mainFunction* element specifies C filename, *renamed\_main.c*, which is used to create the tasks of the application. It is important to note that the *mainFunction* element should not be named '*main*' in order avoid a compilation conflict with SystemC's internal main function. The generated C file creates the tasks from the *task* elements as pthreads. Each *task* is created

as a pthread with the start-routine function and priority as specified by *taskEntry* and *priority* respectively. The *process* element also lists the header and C files which the process and tasks of the main function will use. An example the *process* listed in Listing 3.2 is shown in Figure 3.3.

#### 3.1.1.2.2 Ports

The processor can also have ports which are connected to other components via signals or interfaces. A *port* element includes the name, type and direction of the port. The *name* is used as the variable name for the processor module. The *type* is the variable type used to read or write on the port. The *direction* can be "in" or "out" to distinguish whether the data is being read or written. If a processor has an interrupt controller, its ports are defined in the *interruptController* sub-element. If the processor uses the components via interfaces, there should also be ports with the interface and "*port*" labeled as the type and the direction respectively.

#### 3.1.1.2.3 Interfaces

Interfaces are the API functions to access hardware components. The ports with the *type* set as the user-provided interface class can call these API functions. Implementations of API functions are defined in the hardware components, which the ports are bound to. The interface element needs to define the interface's header file and the interface's functions, return type, and parameters. The functions' name, return type, and parameters are redefined in the global scope to allow the application code to have access to those functions. The redefinition calls the same function though port's interface which links the implementation of the bounded hardware component.



Figure 3.4 shows how the interface function is linked from the application code to the hardware component. The global definition of F1 is used as a wrapper function to call its implementation through the port's variable name, *IF portName* as shown Listing 3.2.

#### 3.1.1.2.4 Interrupt Handler

Another element of the processor is the interrupt handler *interruptHandler*. The interrupt handler manages the interrupt controller's chosen interrupts and calls the appropriate interrupt service routine (ISR). The *name* element is used to create the SystemC thread to represent the interrupt handler. The header specifies the header file which declares the ISR functions. A list of cases also needs to be provided to determine which ISR function to call based on an interrupt signal source. ISR functions are assumed to have a void return and no parameters, thus providing the function name is adequate.

#### 3.1.1.2.5 Interrupt Controller

The last element of the processor is the interrupt controller *interruptController*. The interrupt controller needs a type, name, and list of interrupt ports. The *type* is used as the name of the SystemC module, while the *name* is used as the variable name of the instance at

the system level. The list of interrupt ports, *interruptPort*, initializes all the ports for incoming interrupts with a port name, port type, and port direction. The port type should be Boolean ("bool") because it should only represent an interrupt flag. The port direction is 'in' because the interrupt controller receives the interrupts. The port mapping of the interrupt controller to the processor is not needed because the automation tool implicitly binds them in the generated model.

#### **3.1.1.3** Hardware Components

<hardware> <type>HW1</type> <name>hw1</name> </hardware> <hardware> <type>HW2</type> <name>hw2</name> </hardware>

Listing 3.3: Hardware Components Example

The second set of elements of the system is hardware components. The configuration file may have zero, one, or multiple hardware components, but the definition of the hardware components have to be provided in a header file as SystemC module. Unlike other elements, the hardware element only needs to have a type and a name. Like the processor, the *type* is the module type and the *name* is the variable name. It is also assumed that the file for the hardware is the *type* followed by an ".h" extension. This header file is used by the system file and the Makefile generation. Listing 3.3 shows an example of hardware components in the configuration file. There are two hardware components of *HW1* and *HW2* with variable names *hw1* and *hw2* respectively. The user-provided SystemC modules *HW1* and *HW2* should be available in header files HW1.h and HW2.h.

```
<signal>
<type>bool</type>
<name>interruptSignal</name>
</signal>
<type>int</type>
<name>numberSignal</name>
</signal>
```

## Listing 3.4: Signal Elements Example

The next subset of elements of the system is the collection of signals. Like the hardware components, there can be zero, one or multiple signals declared. A signal needs a type for the signal type and a name for the variable name. Listing 3.4 shows a simple example of two signals; a Boolean type and an integer type with variable names *interruptSignal* and *numberSignal* respectively. The *type* provided should be a defined type in SystemC, C++, or user-defined type. The binding of signals to modules is discussed in the next section.

### 3.1.1.5 Connections

<connection></connection>
<modulename>p</modulename>
<moduleport>interruptPort</moduleport>
<signalbinding>interruptSignal</signalbinding>
<connection></connection>
<modulename>hw1</modulename>
<moduleport>interruptOutput</moduleport>
<signalbinding>interruptSignal</signalbinding>
Listing 3.5: Connection Elements Example

To connect the signals with processors and hardware components, the elements *connection* of the system are used to bind signals and modules. There can be zero, one, or multiple connections. A connection needs the variable names of a module's instance (*moduleName*), its



Figure 3.5: System View of Listing 3.5

port (*modulePort*), and the signal (*signalBinding*). Listing 3.5 shows an example which binds the port *interruptPort* from module *p* to signal *interruptSignal*, and *interruptOutput* from *hw1* to *interruptSignal*. The system layout view of this example is illustrated below in Figure 3.5.

### 3.1.1.6 Run Time

<value>1200</value>	
<sctimeunit>SC_MS</sctimeunit>	

### Listing 3.6: runtime Element Example

The configuration file also defines the logical time until which the SystemC simulation should be run. The *runtime* element is used to set the duration of the run. The elements which define *runtime* are *value* and *scTimeUnit*. The runtime is specified by a number *value* in a time unit *scTimeUnit*. The time unit has to be in SystemC time units which are *SC\_SEC*, *SC\_MS*, *SC\_US*, *SC\_PS*, and *SC\_FS*. They represent seconds, milliseconds, microseconds, nanoseconds, picoseconds, and femtoseconds respectively. Listing 3.6 shows an example that sets up a SystemC simulation to run for 1200 milliseconds.

<timeconsumption></timeconsumption>	
<rtosmodel>QESC</rtosmodel>	
<processorname>p</processorname>	
<sctimeunit>SC MS</sctimeunit>	
Listing 2.7: timeConsumption Element Example	

Listing 3.7: timeConsumption Element Example

The last element of the system is *timeConsumption*. The *timeConsumption* element is used to obtain the busy time of the application running on the processor module's RTOS model. The elements needed for *timeConsumption* are *rtosModel*, *processorName*, and *scTimeUnit*. The *rtosModel* specifies which RTOS model was used and if it is a pre-defined model, the processor's the total consumption time can be easily obtained. *processorName* is the variable name of a defined processor module in the *processor* element. *scTimeUnit* specifies in which time unit the output should be displayed in for the busy and idle time. The time units are *SC\_SEC*, *SC\_MS*, *SC\_VS*, *SC\_PS*, and *SC\_FS* like the *scTimeUnit* in *runtime*. Listing 3.7 shows an example of obtaining the busy and idle time of process *p* with *QESC* as the RTOS model in milliseconds.

## 3.2 Measurement Model Execution Semantics

The cumulative computation of the tasks forms the bulk of the overall computation load offered by the application, since the RTOS overhead is typically very low in streaming applications. For the measurement model, we are only interested in determining the total block delays without any interruptions. The determined delays will be used to model CPU resource consumption of the blocks in the simulation model. SystemC uses a non-preemptive simulation kernel for scheduling its tasks. As such, the application tasks, modeled as SystemC

threads will execute without preemption until they explicitly call a SystemC *wait* statement. We use the *wait* property of SystemC to determine the block delays.



Figure 3.6 Measurement of Application Execution Time on a Processor

Figure 3.6 shows an example of the execution of a measurement model. The application consists of two tasks,  $t_1$  and  $t_2$ , implemented as SystemC threads. We assume that  $t_2$  has a higher priority than  $t_1$ . The application is mapped to a processor (*CPU*), which is implemented as a SystemC module containing the threads. A *Buffer* module models an onchip hardware buffer that generates underflow interrupts to the processor. A SystemC method, which is sensitive to the interrupt signal (*int*), is used to model the interrupt service routine (ISR). The tasks communicate amongst themselves, and with the ISR, by using the communication primitives of the RTOS model.

There are two notions of time in the measurement model: the real time, represented by the wall clock time which is maintained by a free running hardware counter on the target processor; and the logical time of the SystemC kernel. The SystemC time is advanced only by

*wait* statements in the SystemC model of the hardware, which explicitly shows the delays between hardware interrupts. Figure 3.6 illustrates the progress of both real and SystemC time during the execution of the measurement model.

Notable assumptions of this model are as follows. (i) Both real and SystemC times are assumed to be 0 at the reference starting point; and (ii) we assume that a task blocks on receiving a pulse or message. Given these assumptions, the order of execution is as follows:

- (1)  $t_2$  calls RecvPulse(), which consumes  $\Delta_0$  units of real time and is suspended, waiting for pulse from the ISR.
- (2)  $t_1$  executes block A, which consumes 10 units of real time, and calls Recv( $t_2$ ), thereby suspending on message from  $t_2$ .
- (3) Concurrently, the *Buffer* describes the consumption of buffer data by calling *wait* for 5 time units. Therefore, the SystemC time is advanced by 5 units.
- (4) The subsequent underflow interrupt is modeled by setting the interrupt signal, activating the ISR task.
- (5) The SystemC kernel switches context to the ISR method.
- (6) ISR sends a pulse to  $t_2$ .
- (7)  $t_2$  is unblocked, and executes block B, which consumes 5 units of real time.
- (8)  $t_2$  sends a message to  $t_1$  and exits.
- (9)  $t_1$  resumes execution.

The correct simulation of the above scenario would result in a consumption of 10 units of SystemC time by  $t_1$  for block A and 5 units of SystemC time by  $t_2$  for block B as shown in Figure 3.10. In order to compute the above delays, we must annotate code around blocks A and B to check the timer and log the delays. To measure block delays, we have automated the annotation to the application code to identify the blocks. We have automated the

annotation to the application code in different granularities and they will be presented in Chapter 4 and 5.



## 3.3 Logs Processing

**Block Execution Instance** 

Figure 3.7 Logged Data for a Block

After the measurement model has been executed, the logged data for each block is processed. Execution of a block over several iterations may produce very large logs of block delays, but not all logged delays represent only the block code computation. The execution of a block may be influenced by the target processor state, cache state, control flow, DRAM refresh rates, and the scheduling in the base OS. These factors cause the measured delays to vary greatly. Figure 3.7 shows an example of the logged delays for a block over multiple iterations. The smallest delay value (*min*) is the block executed with the best case scenario (high cache hit rates, no context-switching in the base OS, etc.). The *max* delay value shows the biggest recorded delay the block took to execute, which is the worst case scenario. Recall

that the measurement model is executing on a base OS which may preempt a task in the measurement model to perform its own kernel tasks or run other applications outside the model's context. While these scenarios are relatively rare, they produce extremely large block delays, since they include the time for the base OS to complete the preempting task and resume the task being measured. The resulting large delays can skew the processing of these delays and should be treated as outliers. Note that the delays are independent observation of the executed block and that the outliers may occur at any instance. If the outliers are ignored, the remaining delay values between *min* and *actual max* delay value in Figure 3.7 represent the correct number of cycles the block would take to compute, without accounting for the base OS's interference.

#### 3.3.1 Heuristics for Outliers

Since it is impractical to determine exactly which delays resulted from an interruption by the base OS, we use a simple heuristic to filter the delays. We determine the minimum and median delay and consider only those delays that are separated at most (median – minimum) from the median. In other words, we consider only the delays that are less than ((2 \* median) – minimum) because the performance of the application is only of interest at a steady state with a stable cache. The rationale behind the heuristic is that the best case and worst case cache behaviors for a given block execution are likely to be equidistant from the median. It is expected that the minimum value corresponds to the best case cache behavior, while the median value corresponds to the average cache behavior.

#### 3.3.2 Determining an Appropriate Block Delay

In order to meaningfully use the delay information after filtering out the outliers, the remaining logged delays need to be processed. We propose three methods in determining a suitable delay value for each block:

(1) Compute a single average of the remaining delay values:

Add all the delays and divide by the number of delays added.

(2) Compute a single median of the remaining delay values:

Sort the delays from the smallest to the largest value and take the value which lies in the middle. If there is an even number of delays, take the average of the two middlemost values in the sorted delays.

(3) Compute the probability mass function (PMF) of the remaining delay values over a number of bins:

Define equal-sized bins between the minimum and the maximum delay values. We define the probability of the bin for a block is the number of delay values, within the bin, divided by total number of delay values. Then the median of the delay values within a bin is chosen as the representative delay of the bin.

Once a suitable delay (or delay PMF) for each block has been determined, a processed log, which keeps only the determined delays (or delay PMF) for each block, will be generated. If PMF was used to process the delay logs, then the processed log has the probabilities linked with the assigned delays for each bin corresponding to the block.

## 3.4 Simulation Model Structure



Figure 3.8 Simulation Model Generator

The *Simulation Model Generator*, as shown in Figure 3.8, takes the processed log, system configuration, application code, and RTOS model as inputs. It uses these inputs to generate the simulation model which is executed on a host machine to estimate the CPU load offered by the application. Like the Measurement Model Structure, the simulation model generator performs the following steps:

- (1) Parse the *System Configuration* file and create an internal data structure to store the information needed to generate a Simulation Model.
- (2) Generate the SystemC models of the hardware platform, including the processor and peripherals.
- (3) Instantiate the RTOS model in the processor.
- (4) Instantiate the tasks of the applications in SystemC modules corresponding to the processor. The body of each task is filled with calls to the corresponding task entry functions in the application code.
- (5) Modify each file of the application source.

- a. Parse and identify function calls which have a different implementation than the base OS. The identified function calls are replaced by the functions in the RTOS model to disambiguate the functions of the base OS and the modeled target OS.
- b. Do a second parse to annotate measurement code at the specified granularity:
  - i. Identify blocks (basic-block or inter-kernel level).
  - ii. Insert a *START* subroutine/function at the beginning of a block and a *STOP* subroutine/function at the end of the block. The *START* call assigns the starting point to an identifier. The *STOP* call determines the block ID based on the start identifier, and obtains the delay value for the identified block from the processed log. The stop call also annotates the block delay using the RTOS model's time *consume* function.
- (6) Generate the Makefile for the compilation and execution of the output model.



Figure 3.9 Simulation Model on Host Processor

The steps above are almost identical to the measurement model generator. Instead of measuring the time delays, the generated model simulates the time consumption on the RTOS

model. The simulation model is illustrated in Figure 3.9. The processor module and the models of other processors and peripherals are generated in the same manner as the measurement model. Furthermore, the same modifications for ambiguous functions are applied. The annotations in step (5)b remains the same with the exception of the implementations of *START* and *STOP*. The *STOP* subroutine/function uses the consume function of the RTOS, thus advancing the simulation time (timed), to model the delays obtained in the measurement model. The simulation model is executed on the host processor to speed up the time required to estimate the CPU load. The RTOS model now models the delays of the application, and thus advances the SystemC simulation time.

## **3.5** Simulation Model Semantics



Figure 3.10 Block Delay Consumption in Simulation Model

Recall the example in Figure 3.6 in Section 3.2, Blocks A and B were measured to take 10 and 5 units of physical time respectively. The blocks in the generated simulation model will consume the measured time, and hence advance the logical time by calling SystemC *wait* 

statements. The execution of the simulated model is demonstrated in Figure 3.10. At SystemC time 0, task  $t_2$  is suspended on waiting for a pulse from the *ISR*. Therefore, the RTOS model schedules  $t_1$ , which starts executing block A. However, at time 5, the interrupt signal is set by the buffer module, thereby triggering the *ISR*, which calls *SendPulse* method, of the RTOS model to activate  $t_2$ . As a result, the RTOS model changes the state of  $t_2$  to ready, and reschedules the tasks. Since  $t_2$  has a higher priority than  $t_1$ ,  $t_1$  is preempted after executing only 5 units of time of A (represented by sub-block  $\alpha_0$ ). Task  $t_2$  executes block B, consumes 5 units of time to model its delay, and terminates after sending a message to  $t_1$ . At time 10,  $t_1$  resumes and consumes the remaining 5 units of time of A (represented by sub-block  $\alpha_1$ ).

The simulation model uses the *TotalBusyTime* variable in the RTOS model to estimate the total time during which the CPU is busy. The counter simply tabulates all the consumed times for all the tasks during simulation. The *busy time* excludes any time during which all the tasks are suspended, waiting for external hardware interrupts. The total simulated (SystemC) time at the end of simulation model execution is the sum of the estimated total *busy time* and total *idle time*. Hence, the overall computation load offered by the application to the CPU is simply the total busy time divided by the total simulated time.

# **CHAPTER 4**

## **Basic-Block Level Annotation**

In this chapter, we describe how the application source code is annotated at the finegrained basic-block level. To access the basic-block structure, the source code is annotated in assembly, and subsequently used to build the measurement model, to obtain the delay logs. We will also discuss how the basic-block annotated logs can be processed. Finally, we describe the simulation model generator that annotates the assembly code for simulation of the application on target platform.



## 4.1 Application Timing at Basic-Block Level

Figure 4.1 Conditional Branching to Pseudo-Assembly Code

Figure 4.1 shows how a simple if-else condition is transformed to equivalent assembly code. In this example, the if-condition leads to two possible execution paths, A-B-D or A-C-

D. In either case, the assembly code for block A is exectuted first. Then, the if-condition is checked using a comparison instruction, *cmp*, and a conditional jump on zero, *je*. If the condition is satisfied, the assembly code should jump to the *yes* label and execute the code for block C, then fall through to execute the code block D. However, if the condition is not satisfied, the jump to the *yes* label does not occur, and the code for block B is executed. After executing block B, the unconditional jump, *jmp*, goes to the *done* label and executes the code for block D.

Using the dotted lines, it can be seen that path A-B-D takes 200 cycles ([A:100] + [B:50] + [D:50]) and path A-C-D takes 250 cycles ([A:100] + [C:100] + [D:50]) to execute. In order to identify and measure which path was taken, the compiled assembly-level code needs to be annotated.



Figure 4.2 Annotated Pseudo-Assembly Code

## 4.2 Annotation of Application Code

Figure 4.2 illustrates the transformation applied to the sample pseudo-assembly code to identify and measure its blocks. For the full assembly annotation, refer to Appendix B. Calls to the subroutines *STOP* and *START* are added immediately after a label in this order. They are used to identify and measure the basic-blocks. The numerical values in the subroutines' parameters are used for identification. Every time a subroutine is called, the numerical value is incremented, thus making each subroutine call unique. Note that the annotated code

START(ID) begin = ID timer\_start() return STOP(ID)
timer\_stop()
end = ID
delay = timer\_val()
log(begin, end, delay)
return

#### Listing 4.1 START and STOP pseudo-code

introduces additional cycles ( $\Delta t_x$  and  $\Delta t_y$ ) that should not be measured as they are not part of the application nor they are relevant. These additional blocks are shown as hashed bars in the execution path, while the blocks of interest have a solid execution path. Listing 4.1 shows the pseudo-code representation for the *START* and *STOP* subroutines.

The *START* subroutine is used to indicate the starting ID specified by the incremented numerical value, which is saved in *begin*, and to begin measurement. Similarly, the *STOP* subroutine is used to indicate the stopping ID, which is saved in *end*, and to end measurement. The *begin* and *end* variables are used to identify which basic-blocks were executed. Whenever the *STOP* subroutine is called, it associates the *begin* identifier with a corresponding *end* identifier as a block. After identifying a block, the *STOP* subroutine records the number of cycles executed. The following table shows which basic-blocks were computed for the example of Figure 4.2.

Table 4.1 Blocks Identified for Figure 4.2

begin	end	Basic-Blocks	Су	cles
1	2	А	100	= 100
1	4	A – B	100 + 50	= 150
3	4	С	100	= 100
5	6	D	50	= 50

The execution paths of the logged blocks for the two scenarios would be as follows:

no: [1-4] + [5-6] = 150 + 50 = 200 cycles yes: [1-2] + [3-4] + [5-6] = 100 + 100 + 50 = 250 cycles

Recall that the measurement generator uses the RTOS model and the application software uses its API functions. The measurement should not include the execution time of the RTOS model's implemented API functions as they may switch SystemC thread context. The new context may be one of the modeled peripherals, but their time delays should not be included in the predicted CPU load. To omit measurement of the API functions, a new subroutine *STOP\_RTOS\_FUNC*, is used as shown in Listing 4.2. The subroutine's implementation is the same as the *STOP* subroutine in the measurement model.

```
<u>STOP_RTOS_FUNC(ID)</u>
timer_stop()
end = ID
delay = timer_val()
log(begin, end, delay)
return
```

#### Listing 4.2 START\_RTOS\_FUNC Pseudo-code

Although the  $STOP\_RTOS\_FUNC$  executes the same logic as STOP, the measurement model uses the  $STOP\_RTOS\_FUNC$  subroutine to be consistent with the simulation model for block identification. The  $STOP\_RTOS\_FUNC$  subroutine is called before the API function and the START subroutine is called after the API function as shown in Figure 4.3. Using the START, STOP, and  $STOP\_RTOS\_FUNC$  annotation technique for Figure 4.2's example, the execution will measure  $CODE\_A$  as block [1–2] and  $CODE\_B$  with block [3–4] while disregarding the timing delays of the RTOS model's API function.



Figure 4.3 Annotation to Omit Measurement of RTOS Model Functions

## 4.3 Measurement Model Generation

Once the hardware platform model has been generated from the configuration file as described in section 3.1, the generator begins modifying the application source code. The source files are first compiled to obtain the basic-block assembly code. Then the compiled assembly files are passed to an annotation tool to parse and annotate the code as described in the previous section. Once the source code has been annotated, the annotated source files are compiled along with the generated hardware components specified by the configuration to build the executable measurement model. The model then runs on the target machine to obtain a log of all the identified and executed blocks which are specified in the annotation method.



Figure 4.4 Overhead Cycles from Measurement

## 4.4 Logs Processing

The block delay logs, generated from the measurement model execution, are processed as described in section 3.3. Since the annotation was performed at the basic-block granularity, the measured delays for each block are very small (in the order of few hundred hardware cycles). As such, there is a strong likelihood that the measured delay could have been dominated by the measurement overhead (recall the  $\Delta t$  delays from Figure 4.2). The execution times to complete the *START* and *STOP* subroutines are measured as shown in Figure 4.4. The above figure shows only the measurement overhead for the pseudo-code, the full assembly code annotation, shown in Appendix B, has more overhead instructions. Specifically, the {addl \$4, %esp} and {popa} of the START subroutine call and the {pusha} and {push \$#} of the STOP subroutine are measured. Therefore, the logged data should subtract measurement overhead value when filtering out the measured cycles. However, the overhead value is difficult to obtain because the instructions could be pipelined with other instructions. Proper measurements of the overhead will be presented in the Experimental Results chapter section 6.4.1.



Figure 4.5 Distribution of Raw Measured Data of a Block using Basic-Block Annotation

The block delays are processed after removing the measurement overheads from the logged delays. Figure 4.5 shows the distribution of delays for a block in one of our example applications. As discussed in section 3.3.1, the outliers need be removed because the very large delays may be caused by the base OS when it switches context during measurement. Once the data has been filtered (delays smaller than 2 \* median – min), the appropriate delay value can be determined for the block. The basic-block annotated model uses the average or median method to obtain a suitable delay value. The PMF method is not needed for basic-block granularity because the range of the delay values do not vary substantially for the basic-block annotated block.

## 4.5 Simulation Model Generation

The simulation model is generated the same way as the measurement model, with the exception of the implementations of *START*, *STOP*, and *STOP\_RTOS\_FUNC* subroutines. The annotated application code remains the same because the time delays are simulated for the same identified block as the measurement model's source code.

START(ID)	STOP(ID)	STOP_RTOS_FUNC(ID)
begin = ID	end = ID	end = ID
Boolean static firstime = true	delay = getDelay(begin, end)	delay = getDelay(begin, end)
if(firsttime)	Accumulated_delay += delay	Accumulated_delay += delay
{	return	temp = Accumulated_delay
Accumulated_delay = 0		Accumulated_delay = 0
firsttime = false		consume(temp)
}		return
return		

Listing 4.3 Simulation Model Subroutines Pseudo-code

Listing 4.3 shows the pseudo-code of the START, STOP, and STOP RTOS FUNC subroutine for the simulation model. Like the measurement model, the ID parameters of subroutines serve to identify which basic-blocks were executed. The subroutines keep track of the accumulated delays and only exhibit the delay behavior at the STOP RTOS FUNC subroutine. The START subroutine checks if it was the first time START was called, and if so the accumulated delay is reset to 0. STOP obtains a delay for the identified block and accumulates the delay in Accumulated delay. STOP RTOS FUNC obtains the delay for the identified accumulates latest block. it, and consumes the accumulated delays (Accumulated delay). The Accumulated delay is stored temporarily in temp so that Accumulated delay can be reset for the other blocks to execute. The Accumulated delay is reset before the consume call because the consumption may be pre-empted. The accumulated delays are only consumed at the STOP RTOS FUNC to imitate the execution order of the measurement model, which does not call the SystemC wait. The wait is only called by the RTOS model functions, which occurs immediately after STOP RTOS FUNC (not counting the {pusha} and {push \$#} instructions).

# CHAPTER 5

## **Inter-Kernel Level Annotation**

In this chapter, we describe fast and accurate performance modeling using timing delay annotation at the coarse-grained inter-kernel-block level. We will discuss the extraction of inter-kernel delay logs from the automatically generated measurement model, and the processing of the logs. Finally, we describe the simulation model generation and execution to determine application performance on given hardware-software platforms.

## 5.1 Annotation and Generation of Measurement Model



Figure 5.1 Annotation of Measurement Code

In most streaming applications, the delays of primitive computations are dependent, to a much larger degree, on the data size rather than on the data value. Moreover, the primitives usually operate on data frames of fixed sizes. Since the computation along a source path
consists of such primitives, the path delay is data independent. Therefore, we treat the segment source path as an atomic computation block for timing annotation.

Figure 5.1 illustrates the transformation applied to a sample source code to identify and measure its blocks. The *if* condition in Figure 5.1(a) may result in the execution of block (A, B) between kernel calls  $K_1$  and  $K_2$ , or block (A, C) between kernel calls  $K_1$  and  $K_3$ . The annotated code in Figure 5.1(b) is used determine the executed block as well as the delay associated with the block.

The measurement model generator parses the application code and assigns a unique identifier to RTOS kernel calls. As shown in Figure 5.1(b), the generator introduces a variable *begin*, and assigns the kernel call identifier to it, after the call. In this example, the kernel call identifier would be  $K_1$ . It is important to note that  $K_1$ ,  $K_2$ , and  $K_3$  are unique *kernel-call identifiers*; they may or may not be the same kernel function. The model generator adds code to start the time measurement of the block by starting the timer.

The generator also adds code before each kernel call to stop the timer and log the measured time corresponding the executed block. The block is easily identified, since the *begin* variable holds the starting kernel call identifier of each block. The block delay is returned by the function *timer\_val()*. The logged delays for each block are used to compute the probability mass functions to be used by the *Simulation Model Generator*.

### 5.2 Log Processing

Once the measurements of the blocks are logged, the data are filtered and sorted. The execution times of the logged data for each block can vary due to environmental factors described in section 3.3. The measurement overheads are ignored in the inter-kernel annotated

model because the inter-kernel delays tend to be substantially larger than basic-block delays. As such, the measurement overhead is negligible compared to the block delay. The small measurement overheads also occur less frequently because the application is annotated on a much coarser granularity. Therefore, the measurement overheads do not significantly impact the CPU load estimation in the simulation model.

For inter-kernel granularity of annotation, the average, median, and PMF methods can be used to produce the processed log. Figure 5.2 shows the distribution of delays for an interkernel block for one of our example applications. Using the delay distributions, we demonstrate how the PMF of the delays can be generated.



Figure 5.2 Distribution of Raw Measured Data of a Block using Inter-Kernel-Block Annotation

Recall from the basic-block annotation in the previous chapter, not all logged delays are of interest. The outliers (delays larger than 2 \* median – minimum) are filtered out from the data because we want to ignore the large delays incurred by potential interrupts from the base OS.

Since the application code was annotated at inter-kernel level, the filtered data may lie in a larger range of values. Furthermore, there is a greater likelihood of a base OS interruption occurring when measuring the application execution at the coarse grained level.

After filtering, the next step is to sort the filtered data into different bins. The bins are split evenly among the range of filtered block delays. The number of measured data points within the range of each bin is divided by the total number of filtered data points to determine the probability of the bin. The representative value of each bin is determined by obtaining the bin's median delay value. Figure 5.2 shows how the measured data is filtered and split among 10 bins. The probability of each bin corresponds to a relative median value within its set ( $m_0$ ,  $m_1$ ,  $m_2$ , ...), as shown in Figure 5.3. Processing the logs calculates the PMF of every block in the application.



Figure 5.3 Probability Mass Function

#### 5.3 Simulation Model Generator

The Simulation Model Generator takes the processed logs, the system configuration, application code and the RTOS model as inputs. It generates a SystemC model which is

executed on a host machine to estimate the CPU load offered by the application. The application software is re-annotated for inter-kernel identification and for applying a SystemC time consumption function, provided by the RTOS model, to the identified blocks. The blocks' delays are used by the *consume* function to model the CPU time consumption.



Figure 5.4 Block Delay Annotation in Simulation Model

Figure 5.4 illustrates the annotation of the application code during simulation model generation. The simulation model generator parses the application code to identify kernel calls. It introduces a variable *begin* and assigns the identifier of the kernel call to it. The simulation model generator also adds code before each kernel call to *consume* the time for the source path leading to that kernel call. For instance, if the *then* path is taken, the delay for block (A, B) must be consumed. Conversely, if the *else* path is taken, the delay for block (A, C) must be consumed. Furthermore, the delays are consumed stochastically, based on the PMF of the respective block.

In order to model the appropriate delay consumption, the simulation model generator introduces a probability variable p, as shown in Figure 5.4. Before, each kernel call, p is assigned a random real value between 0 and 1, by using the *rand* function. Now, p represents the probability with which we will consume a given delay from the block's PMF. To obtain the actual delay, the range [0, 1] is divided into multiple bins. The number of bins is the same as the number of sets into which the raw measured delays for the block are divided (see Figure 5.2). The size of a bin corresponds to the probability of the delay in the PMF. For instance, for the given PMF in Figure 5.3, we have a total of 10 bins. The size of the bin for  $m_3$  is 0.262, while that for  $m_5$  is 0.1. Over multiple iterations of the block, the value p is expected to be uniformly distributed across the range [0, 1]. As such, over multiple iterations, the delay  $m_3$  will be returned by the *getDelay* function with a probability of 0.262. (see Figure 5.3). Finally, the simulation generator adds code to apply the obtained time delay calling the RTOS model's c*onsume* function.

# **CHAPTER 6**

### **Experimental Results**

In this chapter, we present experimental results for our model generation tools for both the basic-block and inter-kernel granularities. We use three streaming applications: MP3 playback, JPEG encoder, and Voice encoder/decoder. The chosen applications are some of the most commonly used streaming applications on mobile embedded devices. The target platform is QNX RTOS [21], running on a 500 MHz Geode LX embedded processor [22]. The application software, the RTOS model, and the system configuration file are used to generate the measurement and simulation models. We implement a SystemC model of the QNX RTOS scheduler as described in chapter 2. In order to evaluate the quality of our methodology, we measured the three important metrics: the model generation times, the model simulation time, and the timing accuracy of our generated models.

#### 6.1 Use Cases

In order to evaluate our modeling methodology, we focus on four use cases:

(1) For the first experiment, referred to as *MP3*, we simulate only MP3 playback on the phone. The MP3 data is fetched from a file, and the decoded data is written on an on-chip serial buffer. The serial buffer size is set to 32480 bytes, which corresponds to 184.127 milliseconds of decoded stereo audio data at 44.1 KHz. The buffered data is played on the handset speaker.

- (2) The second experiment, referred to as *JPEG*, encodes images taken by phone camera at regular intervals of 250 milliseconds.
- (3) The third experiment, MP3 + JPEG, combines the first two experiments to encode pictures concurrently with MP3 playback. This design uses the same serial buffer size of 184.127 milliseconds and the same image processing interval of 250 milliseconds.
- (4) The fourth experiment, MP3 + Vocoder, performs concurrent MP3 playback and voice encoding /decoding. The encoded voice data of the caller is fetched from the network buffers and decoded. The decoded voice is mixed with the decoded MP3 and written onto the on-chip serial buffer. The serial buffer size is set differently depending on the type of platform design. The different designs have the serial buffer size set to 20, 40, 60, 80, and 100 milliseconds

The first three use cases are evaluated with a fixed platform. The MP3 + Vocoder use case is targeted for platform optimization because it is the most compute-intensive of the four experiments. As we will see in section 6.4, the performance of the MP3 + Vocoder use case can be optimized by modifying the platform.

Figure 6.1 shows the application tasks and the inter-task communication for the MP3 + *Vocoder* application. In the given scenario, the caller wants to play an MP3 clip for the callee, while hearing it on his/her own handset. The audio from the MP3 file must be decoded and mixed with the audio from the phone call at both ends, so that they can speak to each other while the music is playing. Hence, we have four audio streams on the caller's phone:



Figure 6.1 Tasks Sequence and Communication for MP3 + Vocoder

\*MP3: isr, dspaudio, mixerctrl, and audiosal

- Uplink audio: audio being transmitted to the network (including caller's speech mixed with MP3 audio)
- Downlink audio: audio received from the network (the callee's speech)
- Speaker audio: audio being sent to the phone's speaker (includes callee's speech mixed with MP3 audio)
- Microphone audio: audio coming from the phone's microphone (the caller's speech)

For each stream there is a message:

- CLK TX: Interrupt to transmit uplink audio to the network
- CLK\_RX: Interrupt to tell the system that downlink audio has just been received from the network
- I\_AUDIO\_TX\_READY: Interrupt indicating that the D/A converter needs more data to transmit to the speaker
- I\_AUDIO\_RX\_READY: Interrupt indicating that the A/D converter received new data from the microphone.

The embedded software consists of seven tasks: *l1, uas, vocoder, dspaudio, isr, mixerctrl,* and *audiosal.* Task *l1* implements the uplink and downlink events (CLK\_TX and CLK\_RX). Task *isr* is the interrupt handler that notifies the decoding task if more data is needed by the serial buffer for the speakers (I\_AUDIO\_TX\_READY) or if new data is available from the microphone buffer (I\_AUDIO\_RX\_READY). Since the network traffic rates are fixed (as per GSM standards), the only design flexibility is in the size of the speaker's serial buffer. The larger the buffer size, the longer the lag experienced by the listener, from the time that the input audio/voice data arrives.

#### 6.2 Model Generation

		Blocks	Generated	Generated
Application	# of tasks	ID'ed	Lines of Code	Time (s)
MP3	4	433	32765	1.166
JPEG	1	390	13260	2.061
MP3 + JPEG	5	814	40845	2.308
MP3 + Vocoder	7	674	46907	1.225

Table 6.1 Model Generation Time at the Basic-Block Level

Application	# of tasks	Blocks ID'ed	Generated Lines of Code	Generated Time (s)
MP3	4	18	836	1.108
JPEG	1	1	55	2.026
MP3 + JPEG	5	19	277	2.226
MP3 + Vocoder	7	42	338	1.071

Table 6.2 Model Generation Time at Inter-Kernel-Block Level

Table 6.1 and Table 6.2 show the generation time for the annotation of the models for basic-block and inter-kernel annotation, respectively. The number of tasks blocks identified indicates the complexity of the generated model applications. The generated lines of code include the annotated code, for both measurement and simulation, as well as the SystemC model of the platform. MP3 + Vocoder's model generation takes approximately the same amount of time for different platform designs (20, 40, 60, 80, 100 milliseconds of buffer size) because the application code is independent of the hardware parameters. The models for both annotation levels were generated in just a few seconds on an Intel i3 host machine running at 3.20GHz. As we can see, automatic generation of models is much faster and more convenient than the time-consuming, manual annotation at the basic-block level. Therefore, our model generation tools can improve productivity in the design process by drastically diminishing modeling time, and thereby improve efficiency.

#### 6.3 Model Simulation

Another important quality metric for our methodology is the execution speed of the generated models. Clearly, the model execution speed depends on the complexity of the application, the target and host platforms, and the amount of time reserved for measurement or simulation. The complexity of the models depends on the application functionality as well

as the annotations. The generated measurement models are executed on the target Geode LX processor using Linux as the base OS. The generated simulation models are executed on the host Intel i3 machine with Linux as the host OS.

Table 6.3 and Table 6.4 show the measured execution times for the use cases' measurement model, log processing, and simulation model. The total time is considered as the overall time needed to evaluate the design. Note that the processing time of the logs for the inter-kernel annotated models has very little effect (much less than a second) on the overall time and are thus omitted.

Application	Simulated Time (s)	Measurement Time (s) [on target]	Logs Processing Time (s) [on host]	Simulation Time (s) [on host]	Total Time (s)	Total Time / Simulated Time Ratio
MP3	8.0	398.08	356.09	253.46	1007.63	~125 X
JPEG	8.0	431.34	350.91	340.79	1123.04	~140 X
MP3 + JPEG	8.0	819.40	1364.76	774.66	2958.82	~370 X
MP3 + VOCODER (20ms)	0.5	600.33	847.89	476.74	1924.96	~4K X
MP3 + VOCODER (40ms)	0.5	440.12	611.05	352.16	1403.33	~3K X
MP3 + VOCODER (60ms)	0.5	394.76	539.49	311.62	1245.87	~2.5K X
MP3 + VOCODER (80ms)	0.5	367.03	511.67	273.40	1152.11	~2.3K X
MP3 + VOCODER (100ms)	0.5	300.70	406.70	237.77	945.17	~1.9K X

Table 6.3 Execution Speed of Application Models at the Basic-Block Level

Application	Simulated Time (s)	Measurement Time (s) [on target]	Simulation Time (s) [on host]	Total Time (s)	Total Time / Simulated Time Ratio
MP3	300	24.31	5.13	29.44	0.10X
JPEG	300	6.84	0.77	7.61	0.03X
MP3 + JPEG	300	32.66	5.68	38.34	0.13X
MP3 + VOCODER (20ms)	300	222.19	35.15	257.34	0.86X
MP3 + VOCODER (40ms)	300	172.37	27.31	199.68	0.67X
MP3 + VOCODER (60ms)	300	155.48	24.64	180.12	0.60X
MP3 + VOCODER (80ms)	300	146.93	23.76	170.69	0.57X
MP3 + VOCODER (100ms)	300	141.83	22.85	164.68	0.55X

Table 6.4 Execution Speed of Application Models at the Inter-Kernel-Block Level

We notice the following trends:

- (1) The measurement model runs slower than the simulation model since the target platform has a less powerful processor than the host machine.
- (2) Evaluating the inter-kernel annotated models is over three magnitudes  $(10^3 \text{X})$  faster than the basic-block annotated models.
- (3) Increase in buffer size leads to reduction in execution time due to improved cache performance (see Figure 1.1).
- (4) Processing logs is faster with larger buffer sizes. Even though the number of basicblocks in the code is the same, the number times they are executed depends on the buffer size. For example, a smaller buffer size leads to a higher interrupt rate. Hence, the the buffers for inter-task communication will be accessed more often for the same simulated time and the block(s) responsible for buffer access will be executed more often. Consequently, the delay logs for small buffer size will be

longer than the delay logs for large buffer size and the time taken to process the logs is larger for small buffer size.

The basic-block level annotations make the code very complex (see generated lines of code in Table 6.1), and thus the execution time of the measurement model is much higher compared to the simulation times. We use small simulated times for basic-block level measurement in order to reduce the execution times of the models. We record the execution of the generated measurement model on the target processor. This measurement model execution produces the log of basic block delays. The measurement model creates very large delay logs (100–200 MBs for MP3-Vocoder) because the basic-block level annotation identifies blocks, which are executed and measured, at a very fine granularity. The large delay logs are processed on the faster host system. Nevertheless, the processing time is still quite high (as shown on the *Logs Processing Time* column). Finally, we measure the execution time of the simulation model. The execution times from the measurement model, the logs processing, and the simulation model are added and shown in the *Total Time* column.

The final column shows the ratio of total time to the simulated time. The ratios are indicative of the overall simulation performance of our methodology, when using the basic block level annotations. The smaller the ratio, the better is the simulation performance. For instance, a ratio of 125X in the first row implies that it takes 125 seconds to simulate 1 second of software activity on the target hardware. As we can see, this ratio depends on the complexity of the application. However, it is convenient yardstick to measure simulation performance. It can be observed that the overall analysis time (including measurement, log processing, and simulation) is two to three orders of magnitude slower than the simulated time.

The inter-kernel models are much less complex and require much less time to execute on the target and host machine than the basic-block models, because the inter-kernel models are annotated at a coarser granularity. The time taken to process the inter-kernel annotated logs is omitted in Table 6.4 because they are negligible compared to the simulated times. The log files are significantly smaller (19KB–7MB for 300 seconds of simulation for MP3-Vocoder) as opposed to the basic-block level logs (100–200MBs for 0.5 seconds of simulation).

As shown in the last column in Table 6.3 and Table 6.4, the ratio of the total analysis time to the simulated time is much higher for the models annotated at the basic-block-level than for the models annotated at the inter-kernel level. For example, if MP3 were to simulated 100 seconds of computations, the basic-block annotated model would take about 12500 seconds (125X), or about 3.47 hours, to execute and estimate the CPU load. On the other hand, the inter-kernel annotated model would take about 10 seconds (0.1X). Therefore the simulation at the inter-kernel annotation granularity provides several orders of magnitude speed-up over models with basic-block level annotation. However, the accuracy of timing prediction must also be taken into consideration when evaluating a modeling methodology. In the next section, we will discuss the timing accuracy results of our models for the given use cases.

#### 6.4 Accuracy

The most important quality metric for our methodology is the accuracy of the predicted CPU load. In order to make such predictions, measurement overheads need to be considered for the basic-block annotation. In the first section, we perform a short experiment to determine the measurement overhead for the basic-block models. Then, the CPU load is estimated for both the basic-block and inter-kernel annotated models. The different approaches are then compared to determine the most accurate approach.

#### 6.4.1 Measurement Overhead for Basic-Block Annotation



Figure 6.2 Manual Measurement of Overhead

Recall that the subroutines *START* and *STOP* from Chapter 4 cause overhead cycles which may introduce significant errors in the measured delays. Supplemental experiments were conducted on the Geode LX embedded processor to determine the measurement overheads. The *START* and *STOP* subroutines were manually inserted to determine the number of introduced cycles when no relevant code was measured. Figure 6.2 depicts how the overhead cycles were measured. This small experiment always executes and measure block [0–1]. Initially, the experiment measures nothing in between the subroutines. Then the "nop" instructions are incrementally introduced in order to compensate for the effects of pipelining of instructions in the START and STOP routines. The "nop" instruction takes only one cycle to fetch, but does not execute as it is a "no operation". The measurements with the added NOPs are a more accurate representation of the overhead. We performed several experiments by varying the number of NOPs and found that the measurement overhead converged to 100 cycles on the target Geode LX processor.

#### 6.4.2 CPU Load Estimation

Application	Actual	Basic-Block Annotation		Inter-Kernel Annotation		Most
Application	CPU Load	Predicted	Absolute Error	Predicted	Absolute Error	Accurate Annotation
MP3	1.75%	2.81%	1.06%	7.14%	5.39%	Basic-Block
JPEG	2.40%	5.21%	2.81%	2.35%	0.05%	Inter-Kernel
MP3+JPEG	4.15%	11.12%	6.97%	8.22%	4.07%	Inter-Kernel
MP3+Vocoder(20ms)	50.78%	67.75%	16.97%	49.70%	1.08%	Inter-Kernel
MP3+Vocoder(40ms)	38.43%	48.20%	9.77%	37.26%	1.17%	Inter-Kernel
MP3+Vocoder(60ms)	32.81%	43.57%	10.76%	33.19%	0.38%	Inter-Kernel
MP3+Vocoder(80ms)	29.91%	42.09%	12.18%	33.99%	4.08%	Inter-Kernel
MP3+Vocoder(100ms)	28.79%	37.90%	9.11%	33.00%	4.21%	Inter-Kernel

Table 6.5 Estimation Accuracy of Simulation Models Using Average Including Outliers

Table 6.6 Estimation Accuracy of Simulation Models Using Average Without Outliers

Application	Actual	Basic-Block Annotation		Inter-Kernel Annotation		Most Accurate
Аррисации	CPU		Absolute		Absolute	Annotation
	Load	Predicted	Error	Predicted	Error	Annotation
MP3	1.75%	2.11%	0.36%	2.97%	1.22%	Basic-Block
JPEG	2.40%	2.89%	0.49%	2.21%	0.19%	Inter-Kernel
MP3+JPEG	4.15%	4.20%	0.05%	4.31%	0.16%	Basic-Block
MP3+Vocoder(20ms)	50.78%	51.67%	0.89%	47.52%	3.26%	Basic-Block
MP3+Vocoder(40ms)	38.43%	36.38%	2.05%	35.10%	3.33%	Basic-Block
MP3+Vocoder(60ms)	32.81%	31.94%	0.87%	31.05%	1.76%	Basic-Block
MP3+Vocoder(80ms)	29.91%	30.10%	0.19%	29.65%	0.26%	Basic-Block
MP3+Vocoder(100ms)	28.79%	27.40%	1.39%	28.18%	0.61%	Inter-Kernel

Application	Actual	Basic-Block Annotation		Inter-Kernel Annotation		Most
Application	CPU Load	Predicted	Absolute Error	Predicted	Absolute Error	Accurate Annotation
MP3	1.75%	2.19%	0.44%	2.93%	1.18%	Basic-Block
JPEG	2.40%	2.94%	0.54%	2.19%	0.21%	Inter-Kernel
MP3+JPEG	4.15%	4.24%	0.09%	4.25%	0.10%	Basic-Block
MP3+Vocoder(20ms)	50.78%	52.23%	1.45%	53.27%	2.49%	Basic-Block
MP3+Vocoder(40ms)	38.43%	36.67%	1.76%	34.78%	3.65%	Basic-Block
MP3+Vocoder(60ms)	32.81%	32.04%	0.77%	30.74%	2.07%	Basic-Block
MP3+Vocoder(80ms)	29.91%	30.37%	0.46%	29.05%	0.86%	Basic-Block
MP3+Vocoder(100ms)	28.79%	26.60%	2.19%	27.60%	1.19%	Inter-Kernel

Table 6.7 Estimation Accuracy of Simulation Models Using Median Including Outliers

Table 6.8 Estimation Accuracy of Simulation Models Using Median Without Outliers

Application	Actual	Basic-Block Annotation		Inter-k Annot		Most Accurate
Application	CPU Load	Predicted	Absolute Error	Predicted	Absolute Error	Annotation
MP3	1.75%	2.12%	0.37%	2.93%	1.18%	Basic-Block
JPEG	2.40%	2.90%	0.50%	2.19%	0.21%	Inter-Kernel
MP3+JPEG	4.15%	4.15%	0.00%	4.25%	0.10%	Basic-Block
MP3+Vocoder(20ms)	50.78%	51.72%	0.94%	47.07%	3.72%	Basic-Block
MP3+Vocoder(40ms)	38.43%	36.24%	2.19%	34.78%	3.65%	Basic-Block
MP3+Vocoder(60ms)	32.81%	31.57%	1.24%	30.74%	2.07%	Basic-Block
MP3+Vocoder(80ms)	29.91%	30.21%	0.30%	29.05%	0.86%	Basic-Block
MP3+Vocoder(100ms)	28.79%	26.41%	2.38%	27.60%	1.19%	Inter-Kernel

Table 6.5, 6.6, 6.7, and 6.8 show a comparison of the predicted and actual CPU loads for the applications with both the basic-block and inter-kernel annotations. The estimated CPU load is obtained from the simulation model by dividing the RTOS model's total busy time by the simulated time. The actual CPU loads for the reference applications are obtained using the *time* kernel call in QNX, which gives the busy times for the applications during execution on target. Each table processes the delay logs differently. Our goal is to determine whether processing logs as an average or as a median would yield better estimation results. We also want to validate our heuristic of omitting the outliers from the logs.

Table 6.5 predicts the CPU load using all data in the delay logs, without excluding the outliers, and computes an average of all the timing delays for each block. The CPU load estimations obtained in Table 6.6 exclude all the outliers and then calculate an average timing delay for each block. Likewise, Table 6.7 and Table 6.8 process the logs with and without the outliers, respectively, but compute the median delay value, instead of the average, for each block.

As shown in the tables above, most of the simulation models of the applications give a more accurate CPU load prediction for the basic-block annotation without outliers. In general, the computed median delay only considers the data values which are repeated or are in close proximity to one another. Thus, the outlier values have a lesser effect on the median. The average delay is calculated by dividing the total delay by the number of delays. The basic-block annotation poorly predicts the CPU load when average delays with outliers are included because the granularity of the annotation is very sensitive to outliers and measurement overheads. Although we eliminate, the measurement overhead of 100 cycles, this overhead is not a constant and our approximation may lead to errors. However, the other cases estimated the CPU load more accurately than the inter-kernel annotation models, except for *JPEG* and MP3+Vocoder(100ms) applications.

The measurement models which are annotated at inter-kernel level provide a wider range of data for the timing delays of each block. The model's run time consists of several other overheads, such as the kernel activities of the base OS. The actual block delays are recorded in the logs and are processed with the corresponding PMFs.

	Sim.	Predicted	Predicted	Actual CPU	Absolute
	Time (s)	Time (s)	CPU Load	Load	Error
MP3	300	8.88	2.96%	1.75%	1.21%
JPEG	300	6.64	2.21%	2.40%	0.19%
MP3+JPEG	300	12.94	4.31%	4.15%	0.16%
MP3+Vocoder(20ms)	300	142.33	47.44%	50.78%	3.34%
MP3+Vocoder(40ms)	300	105.51	35.17%	38.43%	3.26%
MP3+Vocoder(60ms)	300	93.44	31.15%	32.81%	1.66%
MP3+Vocoder(80ms)	300	89.07	29.69%	29.91%	0.22%
MP3+Vocoder(100ms)	300	84.65	28.22%	28.79%	0.57%

Table 6.9 CPU Load Estimation Using Median delays and 10 bins in the PMF

Table 6.9 shows a comparison of the predicted and actual CPU load. Every block's logged measurements are split into 10 bins to obtain their PMFs. The predicted CPU loads, in the above tables, generate a probability between 0 and 1 for every block to obtain a stochastic delay value from the PMF.

Table 6.10 Average Absolute Errors for Observed Applications Tables 6.5 - 6.8

Logs Processing Type		age of e errors
	Basic- block	Inter- kernel
Average with Outliers	8.70%	2.55%
Average without Outliers	0.79%	1.35%
Median with Outliers	0.96%	1.47%
Median without Outliers	0.99%	1.62%
PMF	N/A	1.33%

Table 6.10 shows the average absolute errors of the tables above. The table summarizes the five methods of processing the logs into a single table. From this table, we can confirm that the basic-block models, with averaging the delay logs without outliers, would give the most accurate CPU load estimation. However, predicting the CPU load with the basic-block models requires much more time. On the other hand, if a fast and reasonably accurate estimation is needed, the inter-kernel annotated model using PMFs would be the most appropriate method.

Despite the relatively high accuracy of our models, our measurements are prone to a few errors. For instance, the cache behavior for the measurement model and the reference design may have some inconsistencies, since the measurement model runs in the context of the SystemC kernel and the base OS. The reference design runs the application on the target OS. Moreover, the state of the processor may be different while executing the same block in the measurement model and in the reference design. The block delay modeling itself may introduce errors, due to its stochastic nature. However, these errors do not have a significant impact as demonstrated in the above tables. Based on this data, we can conclude that our model generators can be used for accurate performance predictions.

#### 6.5 **Design Space Exploration**

Since the MP3 + Vocoder offers a very high CPU load (up to 50%), it is a critical application to be optimized. Figure 6.3 shows that increasing the buffer size reduces the CPU load, but only up to a certain point, beyond which there are diminishing returns on efficiency. Increasing the buffer size for the MP3 + Vocoder case study also increases the delay in the callee's speech being decoded and played back on the caller's speaker. Therefore, the larger the buffer size, the poorer is the quality of service.



Figure 6.3 Buffersize and CPU Load Trade-off

The following tables show the predicted performance improvement for the different serial buffer sizes of MP3+Vocoder. The improvements are obtained from the predicted CPU load from Tables 6.6 to 6.9. For example, Table 6.11 takes the difference between the actual and predicted CPU loads from a buffer size of 20ms to 40ms to observe the performance improvements. So the actual CPU improvement is therefore 12.35% (50.78% – 38.43%). The predicted CPU improvements are the difference between the predicted CPU loads using the same log processing technique.

Log Processing Type	Outliers	Actual CPU Improvement	Basic-Block Inter-Kernel Annotation Annotation Predicted CPU Improvement		Most Accurate Improvement
Average	Include		19.55%	12.44%	Inter-Kernel
Average	Exclude	12.35%	15.29%	12.42%	Inter-Kernel
Median	Include	12.33%	15.56%	18.49%	Basic-Block
Median	Exclude		15.48%	12.29%	Inter-Kernel

Table 6.11 Performance Increase from 20ms to 40ms Buffer Size

Log Processing Type	Outliers	Actual CPU Improvement		Inter-Kernel Annotation ted CPU vement	Most Accurate Improvement
Average	Include		4.63%	4.07%	Basic-Block
Average	Exclude	5.62%	4.43%	4.05%	Basic-Block
Median	Include	5.02%	4.63%	4.03%	Basic-Block
Median	Exclude		4.67%	4.03%	Basic-Block

Table 6.12 Performance Increase from 40ms to 60ms Buffer Size

Table 6.13 Performance Increase from 60ms to 80ms Buffer Size

Log Processing Type	Outliers	Actual CPU Improvement		Inter-Kernel Annotation ted CPU vement	Most Accurate Improvement
Average	Include	2.90%	1.48%	-0.80%	Basic-Block
Average	Exclude		1.85%	1.40%	Basic-Block
Median	Include		1.67%	1.70%	Inter-Kernel
Median	Exclude		1.36%	1.70%	Inter-Kernel

Table 6.14 Performance Increase from 80ms to 100ms Buffer Size

Log Processing Type	Outliers	Actual CPU Improvement	Basic-Block	Inter-Kernel	Most Accurate
			Annotation	Annotation	
			Predicted CPU		Improvement
			Improvement		
Average	Include	1.12%	4.18%	0.99%	Inter-Kernel
Average	Exclude		2.70%	1.47%	Inter-Kernel
Median	Include		3.77%	1.44%	Inter-Kernel
Median	Exclude		3.80%	1.44%	Inter-Kernel

Based on the high accuracy of our simulation models, embedded system designers can investigate trade-offs between quality of service and CPU load for different buffer sizes. For instance, a buffer size of 60ms provides an acceptable quality of service for a tolerable CPU load in the actual design, as shown. If however, quality of service is of primary concern, a buffer size of 20ms can be chosen. In that case, it may be advisable to move some of the compute-intensive functions of the Vocoder to dedicated hardware accelerators instead. The 50.78% CPU load predicted by the model can be used to guide the exploration of hardware accelerated platforms. As seen from the above tables, our modeling methodology supports high speed and reliable design space exploration before a complete software-hardware platform is available.

# **CHAPTER 7**

### **Conclusion and Future work**

In this thesis, we described a methodology and tools to generate accurate and efficient simulation models for streaming applications before the hardware-software architecture is finalized. We implemented measurement at both the inter-kernel and basic-block granularities to provide a detailed analysis of the trade-offs between model execution speed and accuracy. Our results show that the model generation is very fast and the estimated performances are accurate as compared to the target platform. The basic-block level annotation approach provides very accurate performance estimation, but at the cost of slow simulation. On the other hand, the inter-kernel annotation approach provides a very fast and reasonably accurate estimation. The accuracy of our models enables embedded system designer to perform early hardware-software analysis and optimization, before the target hardware and the system software is available. We conclude that a fast and accurate estimation can be obtained using the inter-kernel annotation scheme. However, if estimation time is not a factor, the basic-block level annotation scheme provides more accurate estimation.

Future direction to improve on the modeling techniques presented will focus on multi-core target platforms, improvements to delay processing, and trace analysis. Our models currently support a single-core design. We can extend our modeling technique to multi-core platforms. The major challenge in this respect is the design of a multicore RTOS scheduler. We are likely to see greater variation in execution times of different blocks due to task migration and core-affinity (core preference or assignment of task).

In our current methodology, the block delays are treated independently of one another. The block delay analysis can be improved by finding correlations between block-delays in neighboring blocks. As such, we will need to add a correlation factor to pairs of delays during log processing. Finally, it would be useful for the model to generate a trace of the processor load over time as opposed to a fixed average load. Tracing would allow improved observability of the processor's load, and hence its power consumption, which it is executing different parts of the application. As such, the designer will have better insight into optimization opportunities in the application. In order to model the execution trace of the application of the RTOS model, as well as new functions to display the trace.

# Appendix

#### A. **Configuration File Schema**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- main elements -->
<xs:element name="configuration">
       <xs:complexType>
              <xs:sequence>
                     <xs:elementname="systemcPath" type="xs:string"/>
                     <xs:element name="compiler" type="xs:string"/>
                     <xs:elementname="MESH PATH" type="xs:string"/>
                     <xs:elementref="system"/>
                      <xs:element name="executable" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<xs:element name="system">
       <xs:complexType>
              <xs:sequence>
                     <xs:element ref="processor"/>
                     <xs:element ref="hardware" minOccurs="0" maxOccurs="unbounded"/>
                     <xs:element ref="signal" minOccurs="0" maxOccurs="unbounded"/>
                     <!-- module port binding -->
                     <xs:element ref="connection" minOccurs="0" maxOccurs="unbounded"/>
                     <xs:element ref="runtime"/>
                     <xs:element ref="timeConsumption" minOccurs="0"
maxOccurs="unbounded"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- processor -->
<xs:element name="processor">
       <xs:complexType>
              <xs:sequence>
                     <xs:element name="type" type="xs:string"/>
                     <xs:element name="name" type="xs:string"/>
                     <xs:element name="frequency" type="xs:string"/>
                     <xs:element ref="RTOS"/>
                     <xs:element ref="process" minOccurs="1" maxOccurs="unbounded"/>
                     <xs:element ref="port" type="xs:string" minOccurs="0"
```

```
maxOccurs="unbounded"/>
```

```
<xs:element ref="interface" minOccurs="0" maxOccurs="unbounded"/>
                      <xs:element ref="interruptHandler" minOccurs="0" maxOccurs="1"/>
                      <xs:element ref="interruptController" minOccurs="0" maxOccurs="1"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- interface -->
<xs:element name="interface">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="header" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                      <xs:element ref="function" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- interface functions -->
<xs:element name="function">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="fname" type="xs:string"/>
                      <xs:element name="return" type="xs:string"/>
                      <xs:element name="IF portName" type="xs:string"/>
                      <xs:element name="parameterType" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- RTOS -->
<xs:element name="RTOS">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="type" type="xs:string"/>
                      <xs:element name="model" type="xs:string"/>
                      <xs:element name="name" type="xs:string"/>
                      <xs:element name="library" type="xs:string"/>
                      <xs:elementname="libHome" type="xs:string"/>
                      <xs:element name="policy" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- process -->
<xs:element name="process">
       <xs:complexType>
```

```
<xs:sequence>
                      <xs:element name="name" type="xs:string"/>
                      <xs:element name="header" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                      <xs:element name="cfile" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                      <xs:element name="asIsCfile" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                      <xs:element name="mainFunction" type="xs:string"/>
                      <xs:element ref="task" minOccurs="1" maxOccurs="unbounded"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- task -->
<xs:element name="task">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="taskEntry" type="xs:string"/>
                      <xs:element name="priority" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- port -->
<xs:element name="port">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="name" type="xs:string"/>
                      <xs:element name="type" type="xs:string"/>
                      <xs:element name="direction" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- interruptHandler -->
<xs:element name="interruptHandler">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="name" type="xs:string"/>
                      <xs:element name="header" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                      <xs:element name="cfile" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
                      <xs:element ref="case" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
```

```
<!-- case -->
<xs:element name="case">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="sourcePort" type="xs:string"/>
                      <xs:element name="subroutine" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- interruptController -->
<xs:element name="interruptController">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="type" type="xs:string"/>
                      <xs:element name="name" type="xs:string"/>
                      <xs:element name="interruptPort" type="port" minOccurs="0"
maxOccurs="unbounded"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- hardware -->
<xs:element name="hardware">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="type" type="xs:string"/>
                      <xs:element name="name" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<!-- signal -->
<xs:element name="signal">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="type" type="xs:string"/>
                      <xs:element name="name" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<xs:element name="connection">
       <xs:complexType>
              <xs:sequence>
                      <xs:element name="moduleName" type="xs:string"/>
                      <xs:element name="modulePort" type="xs:string"/>
```

```
<xs:element name="signalBinding" type="xs:string"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<xs:elementname="runtime">
       <xs:complexType>
              <xs:sequence>
                     <xs:element name="value" type="xs:string"/>
                     <xs:element ref="scTimeUnit"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<xs:element name="timeConsumption">
       <xs:complexType>
              <xs:sequence>
                     <xs:element name="rtosModel" type="xs:string"/>
                     <xs:element name="processorName" type="xs:string"/>
                     <xs:element ref="scTimeUnit"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>
<xs:element name="scTimeUnit">
       <xs:simpleType>
              <xs:restriction base="xs:string">
                     <xs:pattern value="SC_(SEC|MS|US|NS|PS|FS)" default="SC_SEC"/>
              </xs:restriction>
       </xs:simpleType>
</xs:element>
```

</xs:schema>

## **B.** Full Assembly Annotation of Figure 4.2

label1:

laber1:			
	pusha		; save all registers into the stack
	push \$	0	; push ID 0 into the stack as a parameter for STOP
	call	_Z4STOP	; correct calling convention to STOP for Intel assembly
	addl	\$4, %esp	; move the stack pointer to skip the pushed 0
	popa		; pop back all registers from the stack
	pusha		; save all registers into the stack
	push \$1		; push ID 1 into the stack as a parameter for START
	call	_Z5START	; correct calling convention to START for Intel assembly
	addl	\$4, %esp	; move the stack pointer to skip the pushed 1
	рора		; pop back all registers from the stack
	; code for block A		
	cmp		
	je	yes	
	; code for block B		
	jmp	done	
yes:			
	pusha		; save all registers into the stack
	push \$2		; push ID 2 into the stack as a parameter for STOP
	call	_Z4STOP	; correct calling convention to STOP for Intel assembly
	addl	\$4, %esp	; move the stack pointer to skip the pushed 2
	рора		; pop back all registers from the stack
	pusha		; save all registers into the stack
	push \$3		; push ID 3 into the stack as a parameter for START
	call	_Z5START	; correct calling convention to START for Intel assembly
	addl	\$4, %esp	; move the stack pointer to skip the pushed 3
	popa		; pop back all registers from the stack
	; code for block C		
done:			
pusha	na ; save all regist		ters into the stack

push \$4	4	; push ID 4 into the stack as a parameter for STOP
call	_Z4STOP	; correct calling convention to STOP for Intel assembly $% \label{eq:stable}$
addl	\$4, %esp	; move the stack pointer to skip the pushed 2

popa		; pop back all registers from the stack
pusha		; save all registers into the stack
push \$	5	; push ID 5 into the stack as a parameter for START
call	_Z5START	; correct calling convention to START for Intel assembly
addl	\$4, %esp	; move the stack pointer to skip the pushed 5
popa		; pop back all registers from the stack
; code	for block D	

label2:

pusha	; save all registers into the stack
push \$6	; push ID 6 into the stack as a parameter for STOP
call _Z4STOP	; correct calling convention to STOP for Intel assembly
addl \$4, %esp	; move the stack pointer to skip the pushed 6
popa	; pop back all registers from the stack
pusha	; save all registers into the stack
push \$7	; push ID 7 into the stack as a parameter for START
call _Z5START	; correct calling convention to START for Intel assembly
addl \$4, %esp	; move the stack pointer to skip the pushed 7
popa	; pop back all registers from the stack

•••

### References

- Frederic Risacher, Kenneth J. Schultz, "Software agnostic approaches to explore pre-silicon system performance," in IEEE High Level Design Validation and Test, 2011.
- Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, Wolfgang Rosenstiel, "Highperformance timing simulation of embedded software," DAC, Jun. 2008.
- Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, "A high-level virtual platform for early MPSoC software development," CODES+ISSS, Sep. 2009.
- 4. OSCI TLM-2.0 user manual, Open SystemC Initiative, 2008.
- Samar Abdi, Yonghyun Hwang, Lochi Yu, Gunar Schirner, Daniel D. Gajski, "Automatic TLM Generation for Early Validation of Multicore Systems," IEEE Design & Test of Computers 28(3): 10-19 (2011).
- Frédéric Pétrot, Marius Gligor, Mian-Muhammed Hamayun, Hao Shen, Nicolas Fournel, Patrice Gerin, "On MPSoC Software Execution at the Transaction Level", *Design & Test of Computers, IEEE*, On page(s): 32 - 43, Volume: 28 Issue: 3, May-June 2011.
- Philipp Kutzer, Jens Gladigau, Christian Haubelt, Jürgen Teich, "Automatic generation of system-level virtual prototypes from streaming application models," *IEEE Symposium on Rapid System Prototyping (RSP), 2011* pp.128-134.
- Andreas Gerstlauer, Haobo Yu, Daniel D. Gajski, "RTOS modeling for system level design," *Design, Automation and Test in Europe(DATE) Conference, 2003*, pp. 130-135.
- Luciano Lavagno, Claudio Passerone, Vishal Shah, Yoshinori Watanabe, "A time slice based scheduler model for system level design", *Design, Automation and Test in Europe, 2005. pp.* 378 - 383.

- Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar, Francisco Blasco, "POSIX modeling in SystemC", Asia and South Pacific Design Automation Conference (ASPDAC), 2006 pp. 24-27.
- Mohamed AbdElSalam Hassar, Keishi Sakanushi, Yoshinori Takeuchi, Masaharu Imai, "RTK-Spec TRON: a simulation model of an ITRON based RTOS kernel in SystemC", *Design, Automation and Test in Europe, 2005, pp.* 554 – 559.
- Zhengting He, Aloysius Mok, Cheng Peng, "Timed RTOS modeling for embedded system design," *Real Time and Embedded Technology and Applications Symposium*, 2005. pp. 448- 45.
- 13. Yonghyun Hwang, Gunar Schirner, Samar Abdi, "Automatic Generation of Cycle-Approximate TLMs with Timed RTOS Model Support", *IESS*, Sept. 2009. pp.66-76.
- 14. Zhonglei Wang, Andreas Herkersdorf, "An Efficient Approach for System-Level Timing Simulation of Compiler-Optimized Embedded Software," DAC, Jul.2009
- 15. Kai-Li Lin, Chen-Kang Lo, Ren-Song Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," ASP-DAC, Jan. 2010.
- 16. Trevor Meyerowitz, Mirko Sauermann, Dominik Langen, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor," DATE, Mar. 2008.
- 17. Yonghyun Hwang, Samar Abdi, Daniel Gajski, "Cycle approximate retargettable performance estimation at the transaction level," DATE, Mar. 2008.
- Parisa Razaghi, Andreas Gerstlauer, "Automatic timing granularity adjustment for host-compiled software simulation," Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific, vol., no., pp.567-572, Jan. 2012.
- Zhonglei Wang, Jorg Henkel, "Fast and Accurate Cache Modeling in Source-Level Simulation of Embedded Software," *Proceedings of Design, Automation & Test in Europe 2013*. pp. 587.
- 20. Kun Lu, Daniel Müller-Gritschneder, Ulf Schlichtmann, "Fast Cache Simulation for Host-Compiled Simulation of Embedded Software," *Proceedings of Design, Automation & Test in Europe 2013*. pp. 637.

- 21. QNX Realtime Operating Systems (RTOS) [online]. Available: http://www.qnx.com.
- 22. AMD Geode<sup>™</sup> LX Processor Family [online]. Available: http://www.amd.com/us/products/embedded/processors/geode-lx/Pages/geode-lx-.processor-family.asp
- 23. Fabrice Bellard, "QEMU: A Fast and Portable Dynamic Translator," USENIX Annual Technical Conference (ATEC 05), Usenix Association, 2005, pp. 41-46.
- 24. SystemC[online]. Available: http://www.accellera.org/.
- 25. SpecC[online]. Available: http://www.cecs.uci.edu/~specc/.
- Felice Balarin, Harry Hsieh, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, A., "Metropolis: an integrated electronic system design environment," Computer, vol.36, no.4, pp.45,52, April 2003
- 27. Richard Lee, Karim Abdel-Khalek, Samar Abdi, Frederic Risacher, "Early system level modeling of real-time applications on embedded platforms," Quality Electronic Design (ISQED), 2013 14th International Symposium on , vol., no., pp.558,565, 4-6 March 2013.
- 28. Ke Yu, Neil C. Audsley, "A Mixed Timing System-Level Embedded Software Modelling and Simulation Approach," International Conference on Embedded Software and Systems. ICESS '09. vol., no., pp.193,200, 25-27 May 2009.
- 29. Kuen-Jong Lee, Chin-Yao Chang, and I-Jou Chen, "EPIDETOX: an ESL Platform for Integrated Circuit Design and Tool Exploration," IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '11)., 381-384.