EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Task execution time prediction for motion control applications

Gozek, A.E.

Award date:
2013

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit
Eindhoven
University of Technology

5T746 - Master's Thesis ES - E

Task Execution Time Prediction for Motion Control Applications

---

A.E. (Emre) Gozek (0786244)
Embedded Systems
A.E.Gozek@student.tue.nl

University Supervisors
dr. ir. J.P.M. (Jeroen) Voeten
J.P.M.Voeten@tue.nl

ir. R.M.W. (Raymond) Frijns
R.M.W.Frijns@tue.nl

ASML Supervisor
N. (Nikola) Gidalov
Nikola.Gidalov@asml.com

October 9, 2013

Contents

Contents **ii**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**
- 1.1 Control Architecture Reference Model - CARM 2G . . . 3
  - 1.1.1 Application Layer . . . 3
  - 1.1.2 Platform Layer . . . 5
  - 1.1.3 Mapping Layer . . . 5
- 1.2 Problem Description . . . 6
- 1.3 Project Contributions . . . 7
- 1.4 Report Organization . . . 7

2 Approach **8**
- 2.1 Task Blocks . . . 9
- 2.2 Experiments on the Real Time Platform . . . 11
  - 2.2.1 Real Time Platform Setup . . . 11
  - 2.2.2 Application Setup . . . 13
- 2.3 Mathematical Model . . . 15
  - 2.3.1 Pure Computation . . . 16
  - 2.3.2 Memory Latency . . . 17
    - 2.3.2.1 Maximum Memory Latency . . . 18
    - 2.3.2.2 Transition in the cache boundary region . . . 18
    - 2.3.2.3 Lower limit and upper limit of cache boundary region . . . 19
    - 2.3.2.4 Calibration of lower limit and upper limit parameters . . . 19
  - 2.3.3 Guide to Use the Model . . . 20

3 Experimental Results and Analysis **21**
- 3.1 Distribution of the Execution Time . . . 23
- 3.2 Comparison of the Predictions with Measurements . . . 25
- 3.3 Accuracy of the Predictions . . . 27
- 3.4 Proof of Feasibility . . . 29

4 Literature Study **30**
- 4.1 Static analysis based methods (Offline) . . . 30
- 4.2 Dynamic information based methods . . . 31
  - 4.2.1 ISS (Low level detailed analysis) . . . 31
  - 4.2.2 High abstraction level approaches . . . 31
  - 4.2.3 Hybrid approach - TLM (Transaction Level Modeling) based approach . . . 32
- 4.3 Comparison of our approach with literature . . . 32

5 Conclusions and Future Work **33**

Bibliography **35**

Appendix **35**

A Terminology **36**

List of Figures

1.1 UV light exposure on the silicon wafer. 1
1.2 Servo control loop with plant 2
1.3 CARM Domain Specific Languages. 3
1.4 Scheduling of tasks in CARM. 5
1.5 Mapping of tasks to the physical platform. 6
1.6 Visualization of the concrete problem description. 7

2.1 Approach that is offered in order to provide a solution. 8
2.2 Approach overview 9
2.3 Parameterization of input parameters by profiling the task blocks. 9
2.4 Approach overview 11
2.5 Scheduling in Linux kernel and Linux kernel with RTAI. 12
2.6 Experiment setup for the application. 13
2.7 Execution time of MAT10x10 block with different sizes of context. 14
2.8 Approach overview 15
2.9 Mathematical Model 16
2.10 Application data alignment in cache line. 18
2.11 Number of cache misses depending on the context size. 19
2.12 Guide to use the model. 20

3.1 Approach overview. 21
3.2 Execution time distribution of MAT10x10 block with context size of 500Kb 23
3.3 Execution time distribution of MAT10x10 block with context size of 7000Kb . . . 23
3.4 Execution time distribution of MAT40x40 block with context size of 7000 Kb . . . 24
3.5 Comparison of model predictions with measurements (MAT10x10). 25
3.6 Comparison of model predictions with measurements (MAT40x40). 25
3.7 Comparison of model predictions with measurements (MAT80x80). 26
3.8 Relative estimation errors for predictions of MAT10x10. 27
3.9 Relative estimation errors for predictions of MAT40x40. 27
3.10 Relative estimation errors for predictions of MAT80x80. 28

List of Tables

1.1 List of chosen task blocks with their features.[3] 4
1.2 FLT_N signal filtering servo control block. 4
1.3 Freescale P4080 Hardware Platform Specification 5

2.1 List of chosen task blocks with their features. 10
2.2 Intel i7 2630qm Specifications. 11
2.3 Chosen parameters of execution platform for modeling. 12
2.4 Input parameters of the mathematical model. 17

Acknowledgment

I would like to express my deepest appreciation to my supervisors Jeroen Voeten, Nikola Gidalov and Raymond Frijns for their great guidance and support throughout my project. Without their guidance and persistent help this thesis would not have been possible.

Moreover, I would like to thank my assessment committee member Ramon Schiffelers and my colleagues at ASML for their support.

Finally, I would like to thank my friends from all around the world and my family for their emotional support.

Abstract

The goal of this graduation project is to predict the execution times of the tasks for motion control applications. These applications are responsible of performing calculations to manipulate the physical entity depending on the data obtained from set point generator and several sensors. While set point generator provides the coordinate information of the next location for a physical entity, actual coordinate information of it is obtained from sensors of the system. Resulting motion commands are transmitted to the plant through the actuators. The time that is spent on this reactive process is specified as IO time delay of the system. ASML's demanding motion control applications introduce challenging IO timing constraints of the system which result in stringent deadlines of the individual tasks.

In ASML's lithography systems, these applications are scheduled on multi-processor multi-core platforms taking into account deadlines of the individual tasks. To this end, the scheduler requires execution time information of these tasks. Currently, this information is obtained by execution time measurement on the target platform. This implies that these measurements have to be carried out for different platforms which are time consuming. In addition, it is difficult to analyze the measurement accuracy because of resource contention and because the measurement itself affects the execution times.

To address these problems this project focuses on accurate model-based execution-time prediction. The level of modeling abstraction should be determined in such a way that a generic solution for different types of applications and platforms is provided. On the other hand, the model should contain sufficient detail to obtain 80-90% prediction accuracy. The model should be able to deal with both multi-core and single-core platforms, implying that the model should not only consider overhead due to private (L1-L2 caches) resource usage, but also consider contention in shared resources (shared memory, off-chip memory and interconnections).

As a result of this project, a generic model is developed which is able to predict execution times of tasks for motion control applications. The proposed model is able to work with any type of task block and single core execution platform by abstracting their essential features. Task execution time predictions with 90% of accuracy are obtained. By using the outcomes of this project, the dependency to the execution time measurement on the target platform can be removed, which can expedite design process of the system. Moreover, obtaining predictions with better accuracy can result in better scheduling in terms of platform utilization.

Chapter 1

Introduction

ASML, as the largest provider of lithography systems for the IC production market, has a crucial responsibility in terms of satisfying the demands for technological improvements of the leading IC producers. The integrated circuit (IC) production process is composed of several steps where photolithography, imaging the patterns of IC design on the wafer, can be considered as one of the most challenging steps. Figure 1.1 visualizes this step where specific areas of the silicon wafer are exposed to UV light by using masks (reticles) on which the circuit design pattern is located. Since silicon wafers are covered by photo-resist material, the areas that are exposed to the UV light can be removed in the further steps and the desired circuit patterns can be formed on the wafer. During the production of ICs, the photo-lithography step is performed several times where in each one of these steps, another layer of the circuit pattern is formed on the wafer and eventually the complete IC is obtained. This operation should be repeated several times to print multiple ICs on the same silicon wafer. This whole process can be realized by either moving the wafer stage (platform that silicon wafer is placed on) and the reticle or changing the direction of UV light source. Since, manipulating the direction of UV light is costly in terms of engineering effort, it is more convenient to keep the light source stable and move the reticle and the wafer according to the patterns that are needed to be imaged. Therefore, ASML lithography systems are designed in a way that multiple components of the illumination and projection module have the ability to move in 6 DoF.

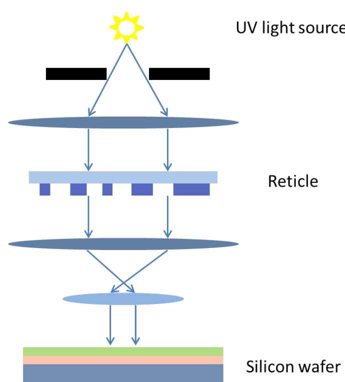

Figure 1.1: UV light exposure on the silicon wafer.

In ASMLs lithography systems, servo control loops are employed which makes it possible to provide the required accuracy and speed for controlling the motion of several modules. Figure 1.2 visualizes the whole reactive control system with its components. It shows that the current position of the plant is obtained from sensors of the system and required calculations are performed in the

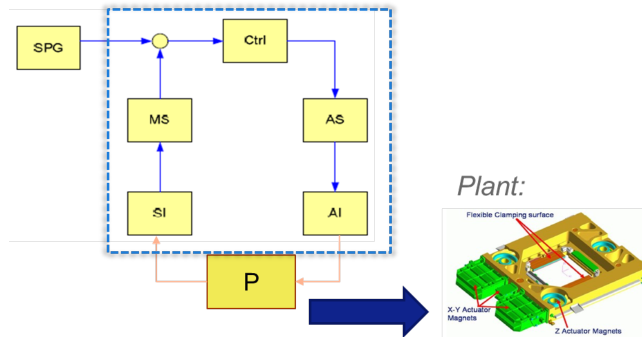

Figure 1.2: Servo control loop with plant

servo control blocks. These calculations are performed by not only considering the next location that is planned to be reached (information obtained from set point generator [SPG]) but also the error value of current location (Difference between the current position and the position that has been intended to be). Finally, the motion control operations are realized on the plants by using the actuators.

According to the Moores law, the size of transistors gates in ICs is expected to be reduced every year, as well as the cost of IC production by taking advantage of technological improvements in IC production process. The IC market has been driven by Moores law for the last decades and the IC market demands to drive this law. In order to satisfy this demand, the throughput of the IC production process and the ability to print finer features on silicon wafers should be improved.

In ASMLs lithoscanners, motion control applications are already compute-intensive with stringent deadlines. By considering the IC markets desire for driving Moores law, these motion control applications are expected to be even more computationally intensive with tighter real-time constraints. Undoubtedly, these demands can be satisfied by employing hardware platforms which can provide higher performance in terms of computational power. The current trend in the IC market indicates that computational power of hardware platforms is increased by employing multi-core platforms due to the limitations of single core platforms.

Multi-core hardware platforms, on which servo control loops can be executed, are already introduced in the CARM 2G project of ASML. However, it is significant to keep in mind that, even if multicore processors can provide more computational power, predicting execution times of the tasks which are mapped to these platforms is more challenging compared to the ones that are executed on single-core platforms. Therefore, scheduling of these tasks in ASMLs lithoscanners, where maximum utilization of the execution platform is aimed and deadline misses of tasks cannot be tolerated, requires accurate predictions for execution times of the task blocks.

The servo control loops are developed in the ASML lithoscanners according to Control Architecture Reference Model (CARM 2G). This model enables specification of the control logic and the execution platform at different levels of abstraction by using domain specific languages (DSL). The development process of servo control loops consists of 3 consecutive steps namely, specification, analysis and synthesis. CARM 2G is explained with its components in more detail, in the following subsection.

1.1 Control Architecture Reference Model - CARM 2G

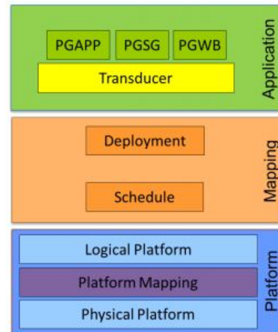

Figure 1.3: CARM Domain Specific Languages.

In order to handle the increasing complexity of motion control applications and hybrid hardware platforms (single core/multi-core multi-processors), it is required to perform analysis early in the design process. Moreover, a seamless parallel multi-disciplinary development process is needed, where deterministic automatic synthesis techniques are employed in order to satisfy both high demands on time-to-market and quality of design requirements.

The CARM 2G framework enables the way of multi-disciplinary working by employing 3 consecutive design process steps namely specification, analysis and synthesis. In the specification step, the system is specified with formal models, using DSLs an analysis step is used to validate the impact of design choices. Finally in the synthesis step, the models that are described in the specification and verified in the analysis are implemented in the real systems. Figure 1.3 shows domain specific languages employed in CARM 2G by classifying them into application, platform and mapping layers. In the scope of this graduation project, among all the languages presented in figure 1.3 task blocks (Application layer - PGWB) and the execution platform (physical platform) are examined in detail. Moreover, in order to understand the scheduling and deployment in CARM 2G, mapping layer is investigated as well. In the following subsections, description and analysis of only the relevant components these layers are provided.

1.1.1 Application Layer

In this subsection, first, descriptions of the task blocks employed in ASML's motion control applications are presented. Afterwards, analysis of the workload of different task block types on the platform by means of memory usage and computation load is provided.

A ***task block*** is the basic computation unit in a servo-group network [2]. It performs a mathematical operation on its inputs to compute its output values. Task blocks are described by means of following entities:

- Input ports are in charge of receiving task block's input values.
- Output ports provide the available results.
- The specification implies the behavior of the block.
- The parameters and properties used to modify the behavior of the block.

The memory load of particular application can be represented by data size of that application.

Table 1.1: List of chosen task blocks with their features.[3]

Block Name	Input Parameters	Parameter type	# of Operations	# of Parameters	Behavior
MAT	M,N	float	(MxN) multiplication+ (NxN) addition	(MxN) + (2xN)	Matrix multiplication for MxN sized matrix
SUMM_F	N	float	(N) addition	N + (2xN)	Summation of N elements (for float)
SUMM_PGD	N	double	(N) addition	N + (2xN)	Summation of N elements (for double)
AND_B	N	boolean	(N) logical 'and'	N + (2xN)	Logical AND operation on N elements
AVERAGE _1F	N	float	(N) addition + (1) division	N + (2xN)	Averages the input signal over N samples

Therefore, the memory load of any task block can be calculated by multiplying the number of variables with the unit size of that variable type. The memory usage term in the context of this graduation project refers to the memory space used on the platform. This memory space can be allocated in different locations on the platform such as private cache, shared cache or off chip memory.

Moreover, the **computational load** of a task block is examined by depending on the number of operations that it needs to perform. During source code compilation, different types of operations can be transformed to different numbers of instructions (each operation type may have a different computational load). Therefore, the computational load cannot be estimated by considering only the number of operations. In fact, the number of instructions (machine code) should be determined to assess the computational load of the task block.

In order to provide a generic model, a common approach to predict the memory usage and the computational load for every different task block type have to be offered. To this end, the variations in the number of variables and the number of operations of each task block is examined by varying its input parameters. Consequently, unique equations are obtained for the number of operations and the number of variables of each different task block type depending on their input parameters. Table 1.1 presents these equations together with some other information for some of the task block types that are examined. In every row of the table, different type of task block is given with its description, input parameters, variable type, number of variables and number of operations.

By varying the input parameter of task block not only the computational load and memory usage but also the implementation (behavior) of it can be changed. Table 1.2 lists different versions of FLT_N (filter function) task block which are in fact derived only by varying one of its input parameters. [3] Since these filter blocks have different implementations, they imply distinct number of operations and number of variables. Therefore, different versions of this task block cannot be represented by single equations. Instead, each different version of task block should be introduced as additional task block type. Consequently, relations between their computational load and memory usage with their input parameters should be derived independently.

Table 1.2: FLT_N signal filtering servo control block.

Block Name	FLT_N
Description	Contains N filters which are connected in series
Configuration	**Behavior**
flt_1order_lp	1st order low-pass filter
flt_1order_hp	1st order high-pass filter
flt_1order_bp	1st order band-pass filter

Table 1.3: Freescale P4080 Hardware Platform Specification

Power Architecture e500 core	
Number of cores	8
Frequency of cores	1.5 GHz (up to)
Sizes of L1 Instruction Caches (Private)	32 Kb
Sizes of L1 Data Caches (Private)	32 Kb
Sizes of L2 Unified Caches (Private)	128 Kb
Size of L3 Cache (Shared)	3 Mb

1.1.2 Platform Layer

The CARM 2G framework defines a platform layer by means of 3 domain-specific languages. The physical platform language contains a description of the hardware and the physical connections as present in lithoscanners. Logical platform language provides the abstracted information from physical platform. Finally, the platform mapping language contains the association information between the logical platform elements and the physical platform entities.

Since this graduation project is planned to provide a generic solution, not only the platform employed in ASMLs lithoscanners, but also architectures of modern multicore platforms are examined and their common characteristics are abstracted. Table 1.3 presents some of the common influential features of these architectures for the platform employed in ASML's lithoscanners. As it is discussed, in the case that the accuracy of the model is lower than aimed, model can be refined by reducing the level of abstraction. In other words, more features can be derived from the platform which are less effective and their effect on execution time can be represented by introducing additional parameters in the model.

1.1.3 Mapping Layer

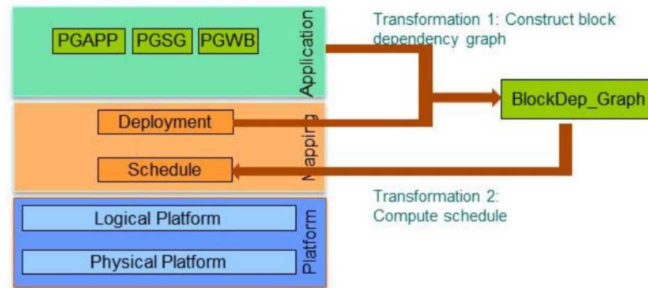

Figure 1.4: Scheduling of tasks in CARM.

The mapping layer in CARM 2G describes the association relation between the task blocks of the application and the processor that these task blocks are going to be executed on. Since a single processor is usually responsible of executing more than one task block, scheduling of these blocks on the processor requires extracting the dependency relation between them. The task block dependency graph can be extracted by applying model to model transformation on servo control loops. Figure 1.4 shows that the latency requirements can be encoded as deadlines for these task blocks after dependency graph is extracted.[6] Moreover, scheduler requires the nominal execution times of task blocks which are provided by execution time measurements on target platform. After the scheduling is performed, the detailed timing analysis of system can be realized to obtain more realistic results [5].

Servo control loops with large number of task blocks are employed in ASMLs motion control applications. These control loops perform computations to manipulate the plant depending on the information received from sensors and set point generator. Eventually, the motions are realized on plant by using actuators of the system. Figure 1.5 visualizes a servo control loop with only few task blocks. It shows the data dependencies between elements (sensors, task blocks, actuators) with straight arrows and mapping of control application entities onto the execution platforms elements with curved arrows.

To conclude this subsection, in order to schedule motion control application tasks, CARM 2G framework requires accurate execution time information of the task blocks. This implies that accurate execution times of task blocks are required to satisfy timing constraints and utilize the execution platform.

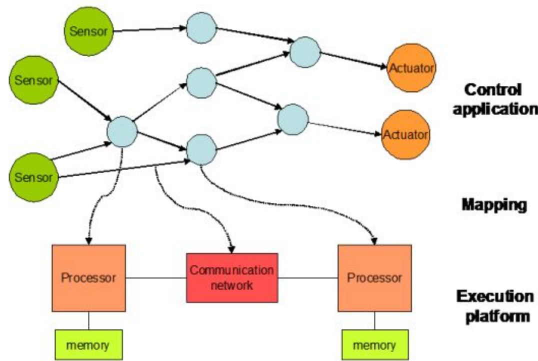

Figure 1.5: Mapping of tasks to the physical platform.

1.2 Problem Description

The main goals of this graduation project are provided below.

- Design a model of the platform and the application in order to predict execution times of task blocks. The level of abstraction should be determined in such a way that a generic solution for different types of applications and platforms is provided. On the other hand, the model should contain sufficient detail to obtain 80-90% prediction accuracy.

Figure 1.6 visualizes the problem description by employing 3 different layers of CARM 2G; application, logical platform and physical platform. The arrow from the physical platform layer to the application mapping step refers extraction of dynamic (run-time) information from physical platform and taking advantage of this information in the application mapping step. Since dynamic information involves communication overhead information within a core, accurate estimations could be obtained. Consequently, scheduling of the task blocks could be realized in a more efficient way (maximize the utilization of the platform).

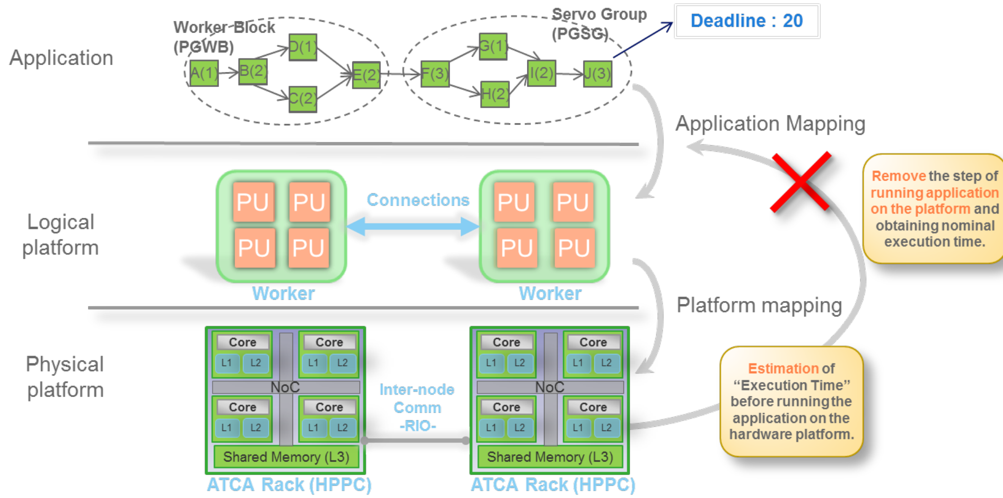

Figure 1.6: Visualization of the concrete problem description.

1.3 Project Contributions

This graduation project provides a flexible generic model which can capture arbitrary task block of motion control applications and any given execution platform. The model is able to predict the execution times of task blocks with relative estimation error less than 20%. The accuracy of the predictions can be tuned by either introducing or hiding some features (modifying the level of abstraction) of the task block and the platform.

As it is mentioned, execution times can be obtained by performing a costly and inaccurate benchmarking procedure, measuring execution times of task blocks. By introducing the model, this step can be removed. As long as the execution time of the task blocks is predicted accurately, the application mapping step in CARM framework can be realized (scheduling and deployment). This model can be extended on any given platform, including single core and multicore for task blocks of ASMLs motion control applications.

1.4 Report Organization

This report is organized as follows. The approach, which is employed throughout the graduation project, is explained in section 2. Experimental results and analysis on these results are provided in section 3. The summary of the literature study that is conducted, is provided in section 4. Finally, section 5 completes the report by providing conclusions and future work of this graduation project.

Chapter 2

Approach

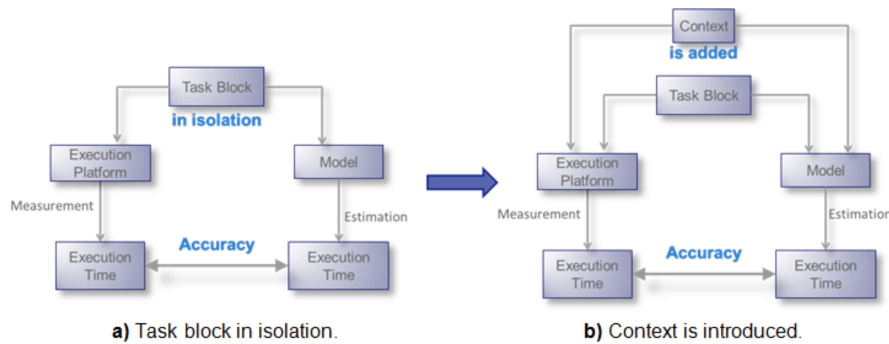

Figure 2.1: Approach that is offered in order to provide a solution.

In this chapter, the approach that is employed throughout the graduation project is presented. Figure 2.1, visualizes two consecutive phases of modeling effort by explaining them in 2 consecutive diagrams (2.1.a, b). First, different task blocks are executed in isolation on the execution platform by varying their input parameters. The model is developed according to measurements that are obtained from these experiments (Figure 2.1.a).

Afterwards, the effect of context (other task blocks that share the same resources) is introduced. Since ASMLs motion control applications contain hundreds of task blocks, a significant amount of computation and memory load can be observed on the execution platform. Therefore, the model should be able to represent the effect of a context, in order to satisfy the accuracy constraints of the predictions. This goal is reached by performing sets of experiments on the execution platform where task blocks are executed with different context sizes. Measurements from these experiments are analyzed by considering the change in execution time. The analysis results are represented in the model by refining it with additional parameters (Figure 2.1.b).

Determining the level of abstraction requires great attention, because the goal of the project (providing generic model with accurate predictions) can be satisfied only by determining this level successfully. We choose an iterative way to set the level of abstraction. Starting from higher abstraction level, the model is refined until the accuracy constraint is satisfied, . The level of abstraction is modified by either introducing or hiding some of the features.

In the following subsections, the modeling process will be explained by two consecutive phases illustrated in figure 2.1.

2.1 Task Blocks

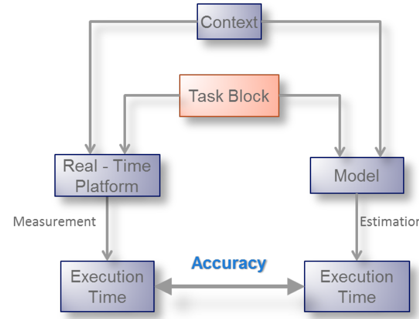

Figure 2.2: Approach overview

This subsection explains how different task blocks of ASMLs motion control applications are analyzed and required information is obtained. Subsequently, how these information used for the model will be provided (Figure 2.2).

In order to gain enough insight regarding the execution time of motion control applications, analysis on different task blocks is performed by examining their implementation. This analysis is performed by using the Valgrind application profiling tool. It executes the application on a virtual platform where parameters of the virtual execution platform can be provided by the user including sizes of different levels of caches, cache line sizes and the associativity of the caches. During the execution of the application on the virtual platform, Valgrind collects dynamic (run time) information and annotates the source code accordingly [12]. Since Valgrind supports instruction set architectures of both experimental execution platforms (Intel i7 2630qm/x86) and target platform (Freescale Power e500mc/Power Architecture), it can be employed for this project.

By employing Valgrind, several different types of dynamic information can be obtained. However, in the scope of this graduation project, 2 types of dynamic information are decided to be investigated namely the number of instructions and the number of cache misses.

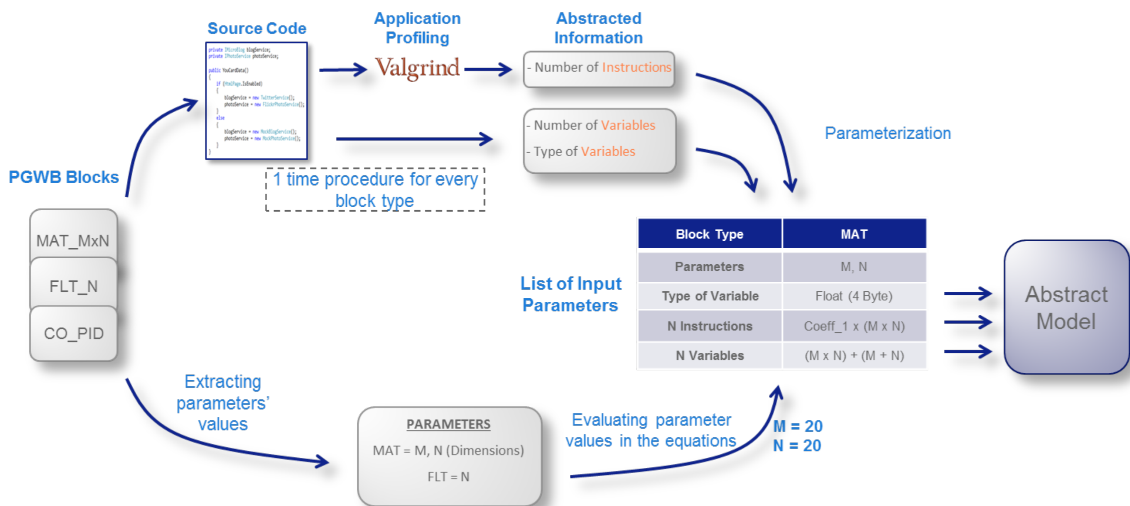

Figure 2.3: Parameterization of input parameters by profiling the task blocks.

Table 2.1: List of chosen task blocks with their features.

Block Name	Input Parameter(s)	Variable type	# of Instructions	# of Variables
MAT (M,N)	M,N	float	Coeff1 x (MxN)	(MxN) + (2xN)
SUMM_F (N)	N	float	Coeff2 x N	N + (2xN)
SUMM_PGD(N)	N	double	Coeff3 x N	N + (2xN)
AND_B(N)	N	boolean	Coeff4 x N	N + (2xN)

Figure 2.3 presents the method to model the task blocks. The upper part of the figure visualizes the parameterization procedure that should be performed once for every block type. We have examined the relation between the number of instructions and input parameters of each task block by varying the input parameters of task block. These experiments are performed by using Valgrind in order to obtain the number of instructions. Consequently, as figure 2.3 illustrates, the number of instructions is parameterized depending on the input parameters of a task block.

Secondly, the relation between input parameters of task blocks and the number of cache misses is examined by using Valgrind. Since the effect of cache misses should be represented in the model without running the application on a target platform or on a virtual platform, we have aimed to predict the number of cache misses depending on the input parameters of task block. The detailed information regarding how to extract this information from the application is presented in section 2.3.2.

In addition to the number of instructions and the number of cache misses, the number of variables used in the task block is analyzed. As it is discussed, this information is required to predict the memory load of the application on the target platform. Figure 2.3 shows that the number of variables can be parameterized depending on the input parameters of task block. This relation can be derived by looking at the source code which are obtained by changing the input parameters of the task block. Naturally, the dependency on third party software is not desired in the final version of the model.

Finally, figure 2.3 presents that variable type is obtained from documentation or even sometimes from the name of the task block itself.

Table 2.1 presents the mathematical equations to predict the number of variables and the number of instructions for the task blocks that are examined. Moreover, it lists the input parameters and the variable type of task blocks. The lower part of the figure 2.3 shows that the mathematical equations that are listed in table 2.1 can be evaluated according to the input parameters of task blocks. Eventually, the number of variables, the number of instructions and the variable type of task block can be provided to the model.

2.2 Experiments on the Real Time Platform

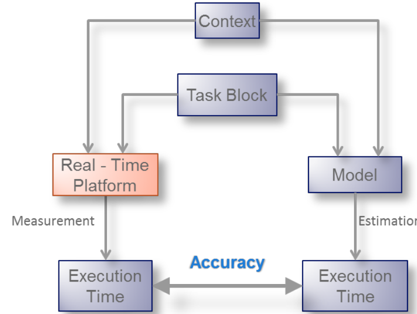

Figure 2.4: Approach overview

In this section, experiments where task blocks are executed on the real time platform, are described (Figure 2.4). Experiments on the execution platform are performed not only for obtaining the execution times from measurements to compare them with predictions and consequently check the accuracy of predictions but also to gain insight on the relation between application size and timing. By using the results of these experiments, the mathematical model is developed. In the model, variations in execution times that are observed from experiments are modeled as accurately as possible.

Since the execution platforms that are employed in ASML's machines are complex and building an experiment setup in these platforms are costly in terms of time and engineering effort, a common general-purpose processor based on "x86" architecture is used for the experiments. One of the main the issues, regarding employing a different execution platform than the actual execution platform, is the architectural differences between these execution platforms (PowerPC vs. Intel/x86). This problem is addressed in section 3.4 in detail.

In the following subsections, the experimental setup is explained in terms of the execution platform and application in detail.

2.2.1 Real Time Platform Setup

Table 2.2: Intel i7 2630qm Specifications.

Number of cores	4
Frequency of core	2.00 GHz
Sizes of L1 Instruction Caches (Private)	32 Kb 8 way associative
Sizes of L1 Data Caches (Private)	32 Kb 8 way associative
Sizes of L2 Unified Caches (Private)	256 Kb 8 way associative
Size of L3 Cache (Shared)	6 MB
Cache Line Size	64 byte

The specifications of the used processor in the experimental setup are shown in table 2.2.

Throughout the project, not only the features of "Intel i7 2630qm" and platform employed in ASMLs lithography systems but also the common features of the modern processor architectures are investigated. The effect of each one of these features on execution times is examined carefully.

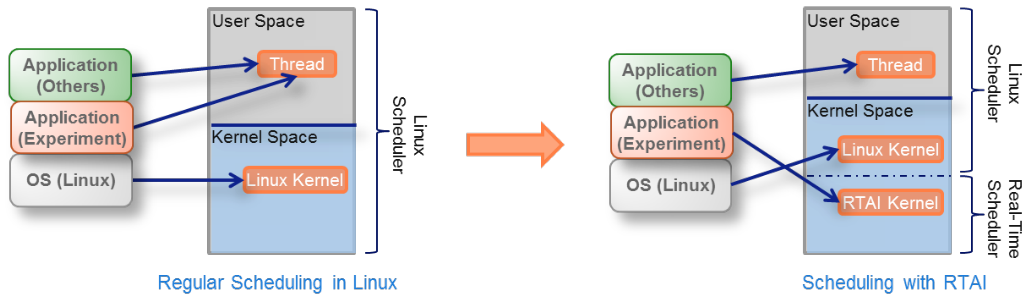

Figure 2.5: Scheduling in Linux kernel and Linux kernel with RTAI.

Depending on their importance either they are modeled as a separate parameter in the model, or their effects are represented in a composite parameter.

In order to provide a generic model with certain accuracy, the parameters in table 2.3 are determined to be employed in the final version of the model. Other features such as cache associativity, cache write protocol, architecture of NoC, TLB cache structure and TLB cache misses are in fact abstracted in some other parameters of the model.

A problem that was experienced in using a general purpose processor instead of a real time platform is the operating systems running on the execution platform. Since the real time platforms are specialized to work without running any operating systems, in these types of platforms it is possible to run applications without any OS based interruption. However, running these experiments on Linux causes variation in execution time of the application due to interruptions of the operating system. Since Linux handles the scheduling in a way that the tasks from its kernel have the highest priority, the execution of task blocks can be interrupted, which may result in inaccurate measurements. Moreover, we have tried to measure the execution time of single task blocks and usually the task blocks that are examined have a small number of instructions. Therefore, it is not possible to ignore the effect of operating system interrupts.

Table 2.3: Chosen parameters of execution platform for modeling.

Abstracted Information from Execution Platform	
Sizes of L1 Data Caches	32 Kb
Sizes of L2 Caches	256 Kb
Size of L3 Cache	6 MB
Cache Line Size	64 byte

In order to eliminate the effect of the operating system based interruptions, RTAI (Real-time application interface) is employed. It is a real-time extension for the Linux kernel which allows users to implement applications with strict timing constraints for Linux [13].

By introducing RTAI, a new layer is defined under the Linux kernel and applications that are mapped to this layer are scheduled with the highest priority. In this way execution of task blocks in our application are completed without experiencing any OS based interruption. Figure 2.5 presents how the scheduling of our application is modified where the diagram on the left side refers to the regular scheduling scheme of Linux and the one in the right side visualizes the way of scheduling with RTAI.

2.2.2 Application Setup

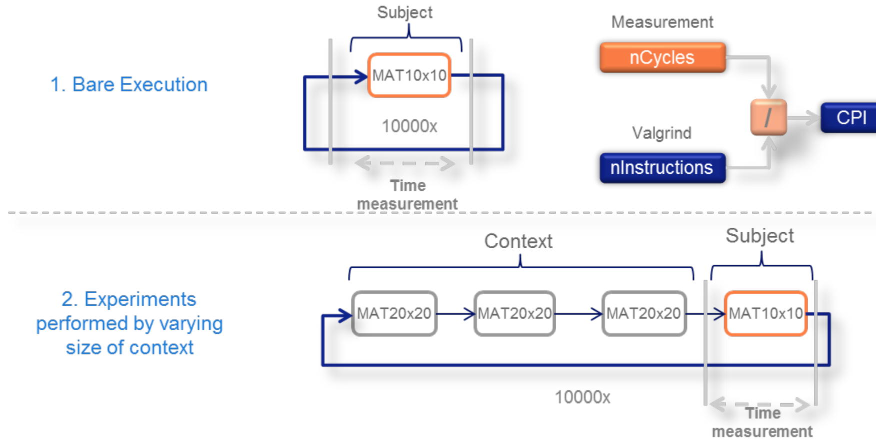

Figure 2.6: Experiment setup for the application.

As mentioned earlier in the report, the model should be able to predict the execution time not only considering task blocks themselves but also the effects of the application that our task block belongs to (context). In order to examine the effect of the context size, first, task blocks are executed alone on the platform. Afterwards, they are executed by varying the sizes of context on the same processing unit. Subsequently, the changes in the execution times of the single task blocks are examined.

The upper part of figure 2.6 illustrates how of single task block (bare execution) is executed and how CPI value is determined by using measurements from this experiment. In this particular example, 'MAT10x10' block is chosen as the task block to examine (subject) and its average execution time is obtained from running it '10000' times.

The average execution time is determined in terms of number of cycles and it is divided by the number of instructions of task block to provide CPI value. We have used CPI parameter in the model to abstract the effect of all features of the CPU other than latency due to L2-L3 and off-chip memory usage. Since CPI represents the average speed of the execution platform for each particular task block, it has to be determined for every different task block individually. This parameter is used to calculate execution time except memory latencies by multiplying it with the number of instructions. Considering the fact that, number of instructions is relatively large value and memory latency is usually responsible of smaller part of whole execution time, CPI is one of the most influential parameters of the model. Therefore, accuracy of the execution time predictions strongly depend on CPI value.

Apart from bare execution, figure 2.6 shows that task block is also run several times with different context sizes (starting from 100 Kb to 7500 Kb). In each experiment, the task block is executed 10000 times in order to average out anomalies and to obtain execution time distribution. By analyzing distribution graph, we can asses the predictability of the execution platform.

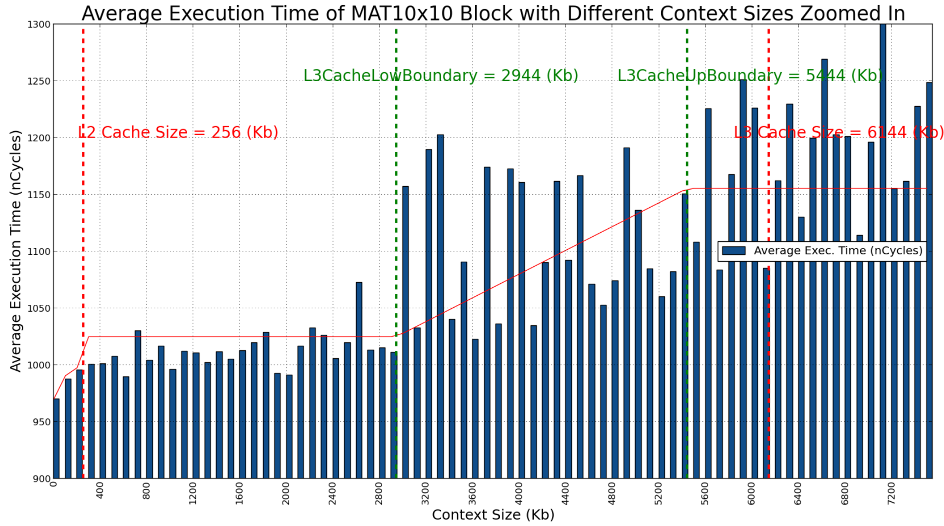

Figure 2.7: Execution time of MAT10x10 block with different sizes of context.

Figure 2.7 shows the variation in the execution time of a MAT10x10 block depending on the size of the context that it belong to. Moreover, the red line in figure 2.7 refers to trend line of the blue bars (blue bars refers to the average execution time of the task block with that particular context size). The behavior of execution time can be derived from this trend line. As it can be observed from figure 2.7, the execution time has a negligible variation until the block starts to suffer from L3 cache misses. Starting from a certain size of context, there is an increasing trend in execution time until the context size approaches the L3 cache size. Once the context size exceeds the cache size (in the rightest part of the figure 2.6) the execution time saturates around a certain value which can be explained by the fact that maximum number of cache misses is reached.

The green dotted lines in figure 2.7 refer to the starting and ending points of the specific region. In the context of this graduation project, we call this region the cache boundary region. In this region, the transition between two different values of the execution time can be observed. This transition is mimicked in model by using a linear transition.

To provide accurate task execution time predictions for all different context sizes, upper and lower limits of the cache boundary region should be determined as accurately as possible. Section 2.3.2.2 addresses the approach on how to determine these values in detail.

2.3 Mathematical Model

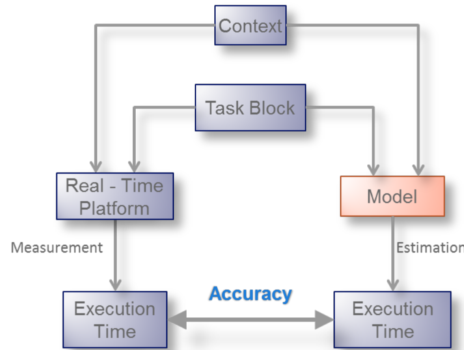

Figure 2.8: Approach overview

The model should be able to predict execution times of task blocks by considering the effect of context (The application that a task block belongs to). Therefore, first we try to obtain accurate prediction for the task block in isolation. Later, the effect of different context sizes is introduced by updating the model with new input parameters. Subsequently, the accuracy of the model predictions is examined.

During the literature study of this graduation project, the effects of different features of modern processor architectures on execution time are examined. Similarly, relevant features of the applications are investigated with their effect on execution time. Since the goal of this project is to provide a model where the level of abstraction is kept as high as possible, we have started to develop the model by introducing few parameters. Consequently, accuracy is increased by iteratively reducing the level of abstraction.

Since figure 2.7 presents execution times of a task block where wide range of a different context sizes are mapped to an execution platform, information regarding not only task block execution time for a specific context size but also how execution time changes depending on the context size is derived from the figure. Therefore, the mathematical model presented in this section is developed based on the analysis of the execution time observed in figure 2.7.

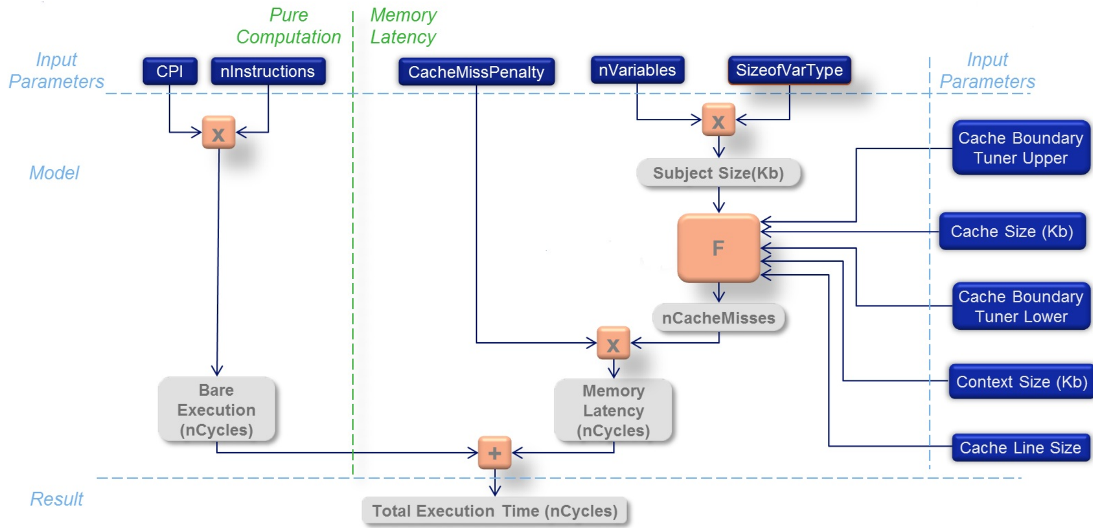

Figure 2.9: Mathematical Model

Figure 2.9 presents the final version of the mathematical model that is developed. In the left part of the green dotted line, the calculation of the pure computation time is visualized. The right part of the green line is in charge of predicting only the memory latency. The total execution time of the task block is obtained by summing up these values. Several input parameters are used to predict the execution time accurately. Table 2.4 presents not only the definitions of these input parameters but also the way to obtain the values of these parameters from the application and the execution platform.

2.3.1 Pure Computation

Figure 2.9 shows that in order to obtain pure computation time, CPI is multiplied with the number of instructions. Number of instructions of task block is obtained by evaluating the equations given in table 2.1 with values of input parameters of that task block.

As it is discussed, the CPI (Cycles Per Instruction) is the key parameter that captures all different features of the current platform and the application, in case no cache misses are experienced. Therefore, having an accurate CPI value is required to predict the execution time accurately. The CPI value is obtained by executing a task block without context (in isolation) as it is visualized in figure 2.6.

It is significant to state that, the pure computation time forms the largest part of the total execution time. Therefore, accurate execution time predictions can be provided if and only if the pure computation time is estimated accurately. Moreover, current version of the model predicts the pure computation time only by using two different input parameters (CPI and number of instructions). This approach makes the CPI parameter extremely sensitive and therefore it has to be obtained accurately for all different task blocks. In fact, experiments on the "MAT_MxN" block showed that same task blocks with different input parameters have similar CPI values. However, even the relatively small variation in CPI has significant effect on the pure computation time. Thus, the final version of the model requires extraction of the CPI parameter for each different task block.

2.3.2 Memory Latency

The right part of the green dotted line of figure 2.9 shows how memory latency is calculated by depending on its relevant input parameters. Multiplication of the number of cache misses with the cache miss penalty memory latency provides the memory latency. Since the number of cache misses of a task block is a dynamic (run-time) information and it depends on several parameters of both the execution platform and the application, its prediction is not straightforward.

Function "F" provides the number of cache misses depending on the subject size (size of the target task block), context size, cache line size and cache boundary region. Figure 2.9 visualizes that the subject size can be calculated by multiplying the number of variables that are used in that task block with the size of the variable type.

In the scope of this graduation project memory latency is examined by considering 3 different regions observed in figure 2.7. In the first region (left part of cache boundary region's lower limit) no memory latency (due to L3 cache misses) is experienced until the application data doesn't fit in the L3 cache memory anymore. In other words for that particular region, number of L3 cache misses is observed as 0. Figure 2.7 also shows that the execution time transits from a lower level to higher level in a certain region whose boundaries is marked with green dotted lines. Finally, in the third region, the memory latency saturates to a certain value where the summation of context and subject size exceeds the upper limit of the L3 cache boundary. Since there is a certain maximum value for number of cache misses of a task block, memory latency that can be caused by a task block has a maximum value. In the following subsections, how the proposed model predicts the memory latency is presented by considering the three regions defined above.

Table 2.4: Input parameters of the mathematical model.

Name of Input Parameter	Definition / Method
CPI (Cycles per Instructions)	Average number of cycles required to execute 1 instruction.
	CPI value is obtained from the execution of task block in isolation
nInstructions	Number of instruction (lines of assembly code).
	Obtained by evaluating the equation provided in Table 2.1. Note that the equations in table 2.1 provided by application profiling (Valgrind).
CacheMissPenalty	Number of cycles needed to retrieve data from a higher level of memory.
	Can be obtained either from datasheet or by using hardware calibrator software (LMBenchmark, HardwareCalibrator [12]).
nVariables	Number of variables that task block contains.
	Can be obtained by analyzing the source code.
SizeofVarType	Size of the dominant variable type used in the task block (Float 4 bytes).
	Can be obtained from documentation of the task block.
CacheSize	Size of the different levels of cache.
	Datasheet of the execution platform provides size for different levels of cache.
ContextSize	Size of the application that the task block belongs to.
	nInstances x nVariables x SizeOfVariableType
Cache Line Size	Size of the cache lines of the execution platform.
	Datasheet provides cache line size information.
Cache Boundary Region Limits	Refer to points at which a task block starts to experience cache misses and where the number of cache misses saturates at its maximum value.
	They are defined as tunable parameters and can be calibrated(Section 2.3.2.4).

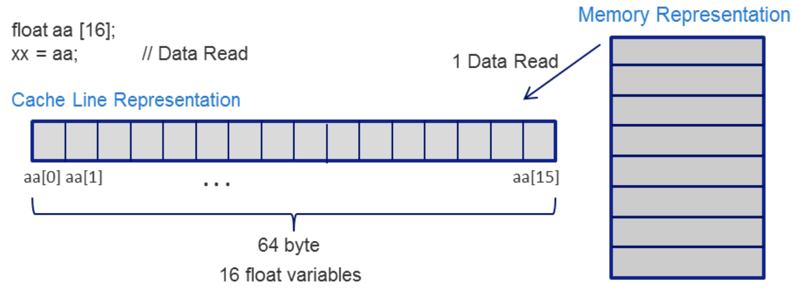

Figure 2.10: Application data alignment in cache line.

2.3.2.1 Maximum Memory Latency

The maximum number of cache misses is calculated according to the following assumption. Every cache line that our application data is mapped to is fully utilized. As it can be observed from figure 2.10, it is possible to fit 16 float type variables in a single cache line of 64 bytes. Since in every cache misses that the execution platform experiences it retrieves data from one whole cache line, maximum number of cache misses that our application can experience is the number of cache lines that application data is mapped to. Therefore, by dividing the subject size with the cache line size, the number of required cache lines to map our task block can be determined. The approach employed in our model to calculate maximum memory latency can be justified by following two facts. First, task block implementations are optimized for code utilization in order to meet real-time constraints. Therefore, in these implementations cache lines are utilized as much as possible. Secondly, the idea of providing a generic model requires keeping the level of abstraction as high as possible and consequently dealing with as less as detail possible. Since the maximum number of cache misses can be calculated easily in this way without introducing significant inaccuracy in predictions, this approach is employed.

2.3.2.2 Transition in the cache boundary region

In order to predict the number of cache misses in the cache boundary region, the behavior observed in figure 2.7 is tried to be represented in the model as accurately as possible. Linear transition is applied between lower limit and upper limit of the cache boundary region from "0" cache misses to maximum number of cache misses. This approach is visualized in figure 2.11, where in the upper part of figure, the graph provides the number of cache misses depending on the context size. Moreover, in the middle part of the figure, memory (i.e. L3 cache) and alignment of the context and subject data in the memory is visualized. Finally, the lower part of figure 2.11 illustrates the amount of application data that could be fit to the memory. Accordingly, the unfit data is visualized which determines the number of cache misses in the cache boundary region.

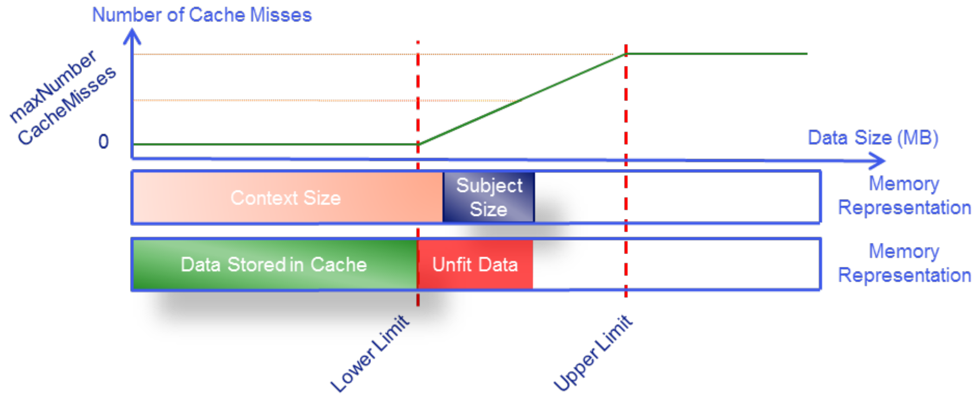

Figure 2.11: Number of cache misses depending on the context size.

2.3.2.3 Lower limit and upper limit of cache boundary region

Figure 2.7 shows that transition of task block execution time between no memory latency and the maximum memory latency occurs in a certain region of application data. To represent the region of transition, the cache boundary region is introduced to our model with its lower and upper limits. The memory utilization at which a task block starts to experience cache misses and at which it saturates is examined to determine the lower and the upper limits of the cache boundary region. In the next subsection the way that lower and upper limit of cache boundary region is presented. By using a linear approximation the number of cache misses within this range can be predicted.

2.3.2.4 Calibration of lower limit and upper limit parameters

Lower and upper limits of cache boundary region depend on how efficiently the application data is aligned on the cache lines. Moreover, utilization of cache lines strongly depends on the type of the application and how efficiently memory is utilized in the implementation of the application. Therefore, we have decided to define these parameters as tunable parameters. By calibrating the model depending on the application, these parameters can be set to minimize the average relative estimation error of predictions.

Throughout the project, these parameters are calculated by assigning several different values to them in a certain range in an iterative way. For every assigned value the accuracy of the prediction is calculated. Consequently, values that provide the minimum average relative estimation error are used as lower and upper limit of cache boundary region.

After lower and upper limit of the cache boundary region is determined by calibrating the model, number of cache misses of the task block can be predicted depending on the context size. By multiplying the number of cache misses with the cache miss penalty, the memory latency can be predicted. Finally, by summing up memory latency and bare execution time, total execution time can be provided in terms on number of cycles as it is visualized in figure 2.9.

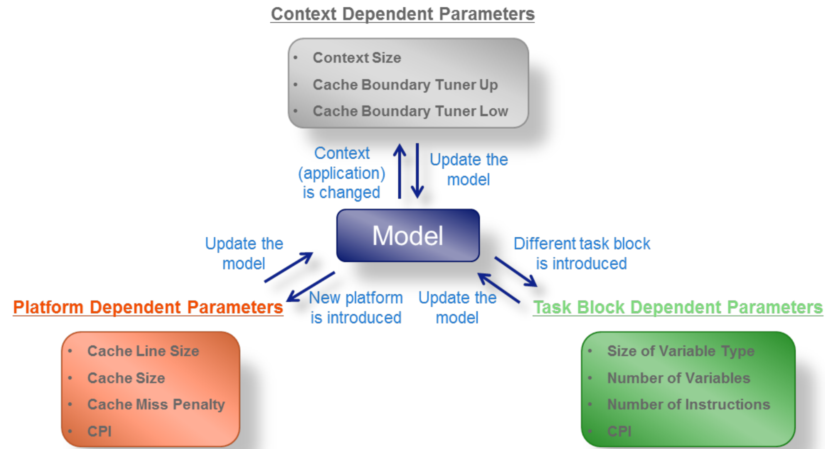

Figure 2.12: Guide to use the model.

2.3.3 Guide to Use the Model

Figure 2.12 illustrates the guide to use the model, where the parameters of the model that requires calibration when platform, task block or context is changed. Platform dependent parameters including cache line size, cache size, cache miss penalty and CPI is expected to be re-calibrated whenever a new execution platform is introduced to the model. Similarly, in case execution time of a new task block type is aimed to be predicted, task block dependent parameters should be updated. These parameters are size of variable type, number of variables, number of instructions and CPI. Finally, whenever the application is changed, the context size should be updated. Moreover, calibration has to be performed to recalculate the lower and the upper limits of the cache boundary region.

Chapter 3

Experimental Results and Analysis

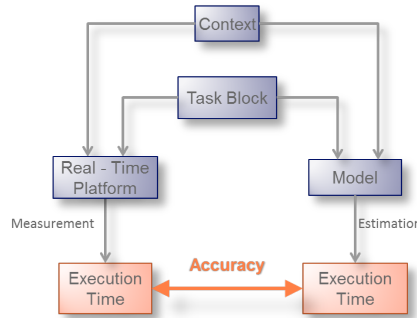

Figure 3.1: Approach overview.

In this section, comparison of the predictions with the measurements is presented (Figure 3.1) in order to confirm that the accuracy constraint of the project is satisfied. Moreover, analysis on execution time distribution graphs of task blocks is provided to give an idea of the level of predictability of the execution platform.

ASMLs motion control applications consist of hundreds of task blocks with strict timing constraints. These task blocks are mapped on the execution platform to satisfy these constraints and utilize the execution platforms as much as possible. Therefore, these execution platforms often used in their limits by means of resources (Private and shared memories). By considering these facts, we have to verify that the model can provide predictions with certain accuracy in every possible scenario. To this end, experiments are performed on MAT10x10, MAT40x40 and MAT80x80 task blocks by varying the context size between 0 Kb and 7500 Kb. Moreover, each experiment setup is executed 10000 times in a for-loop fashion in order to not only eliminate the anomalies from measurements but also presenting the distribution of execution times. These distribution graphs can be used to analyze the predictability of execution platform and application.

In the following subsections, 3 different types of experiment results are presented. Each one of them addresses different concern of the model. Consequently, analysis and interpretation of these results are provided. The types of graphs, with the reason that they are chosen, presented as follows:

1. *Distribution of execution times* for MAT10x10 and MAT40x40 blocks with different context sizes.
   Features:
   - These graphs are prepared in a histogram fashion.
   - In these graphs, predictions from the model and the relative estimation errors according to these predictions are indicated as well.

   Reasoning:
   - Analysis of the execution time distribution graphs for the same task block with different context size shows the effect of context size on predictability of execution time.
   - Comparison of distribution graphs of different task blocks (MAT10x10, MAT40x40) with the same context size shows the relation between predictability and the size of the task block.

2. *Comparison of predictions and average execution times* for different context sizes.
   Features:
   - In these graphs, L2 and L3 cache sizes are presented as well as the lower and upper limit of the cache boundary region.

   Reasoning:
   - By comparing the predictions with the measurements, it is possible to learn how accurate the predictions are. Unlike relative estimation error graphs, these graphs can present the information regarding execution times are underestimated or overestimated.
   - By analyzing the start and end point of the increasing trend in execution time (cache boundary region), we can observe how successfully the model is calibrated for lower and upper limit parameters.

3. *Relative estimation errors* for the task block with different context sizes.
   Features:
   - In addition to relative estimation errors, L2 and L3 cache sizes are illustrated in the graphs as well.

   Reasoning:
   - The relation between accuracy of the predictions and size of context can be analyzed.
   - Similarly, by comparing the relative estimation graphs of different task blocks, the relation between size of task block and accuracy of the predictions can be derived.
   - In case inaccurate predictions are obtained, we can diagnose the problem in the model by analyzing the location of inaccurate results. For instance, if the inaccurate results are experienced after L3 cache misses are introduced to the model, problem can be caused by faulty L3 cache miss penalty.

3.1 Distribution of the Execution Time

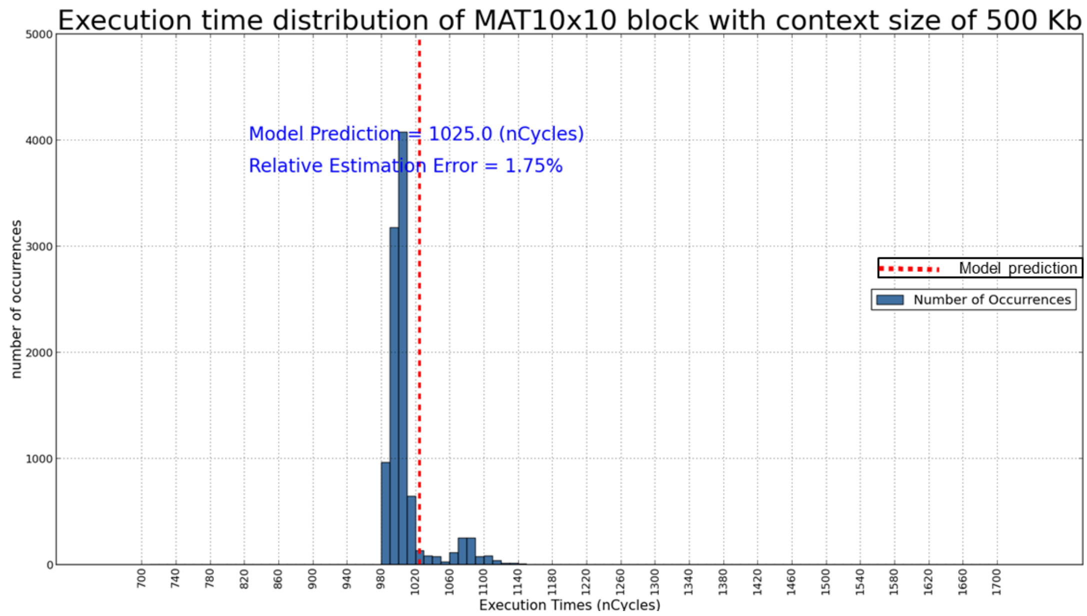

Figure 3.2: Execution time distribution of MAT10x10 block with context size of 500Kb

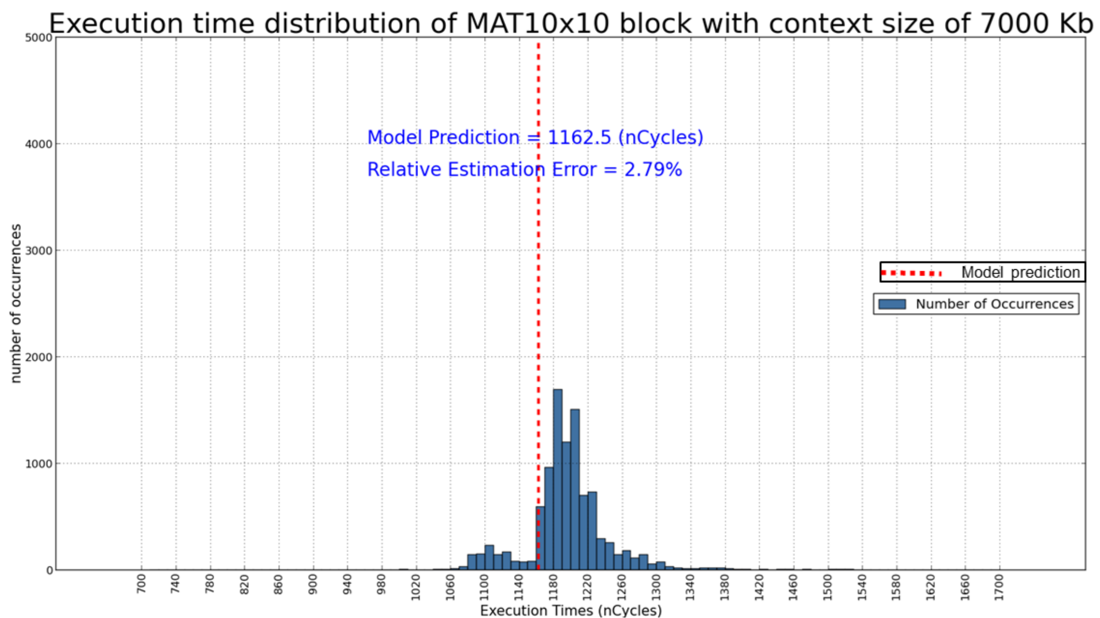

Figure 3.3: Execution time distribution of MAT10x10 block with context size of 7000Kb

In figure 3.2, execution time distribution graph for MAT10x10 block is presented where context size is given as 500 Kb. Similarly, figure 3.3 provides the execution time distribution graph of the same task block, but this time with 7000 Kb of context size. In addition to the distributions of the execution times (blue bars), predictions that are provided by model are shown as red dotted lines in both figures.

Figure 3.2 presents a left skewed histogram graph which can be explained by the fact that there is a certain minimum value that the platform can complete execution of this task block. It also shows that the execution platform provides predictable behavior in case no L3 cache misses are experienced.

Comparison between figure 3.2 and 3.3 shows that the histogram bars are slightly shifted to the right side of the graph. In other words, execution time is likely to be larger when larger context size is employed. This observation verifies that larger memory latency is experienced when 7000 Kb of context size is mapped on the execution platform. In both cases, the model is able to provide predictions which satisfy the accuracy constraint of the project.

Furthermore, another outcome of the comparison between figure 3.2 and 3.3 is the growth in the width of the histogram. This observation implies less predictable behavior of the execution platform in case context size is larger. Since modern processor architectures provide several mechanisms to hide the memory latencies, a processor is able to complete execution in a smaller amount of time. However, these speculative execution mechanisms reduce the predictability of the platform as can be seen in figure 3.3 and 3.4.

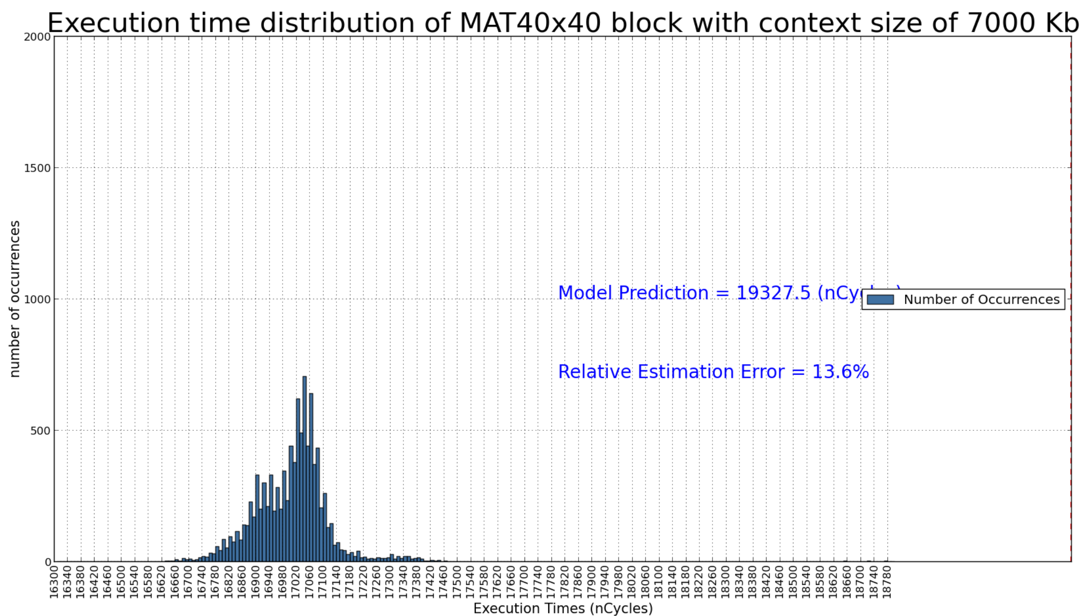

Figure 3.4: Execution time distribution of MAT40x40 block with context size of 7000 Kb

Finally, comparison of figure 3.3 and 3.4 shows that width of execution time distribution is increased when larger task block is executed with same context size. Since a larger task block has a larger number of instructions and it requires a larger number of data access, the speculative execution mechanisms causes more variation in execution time.

3.2 Comparison of the Predictions with Measurements

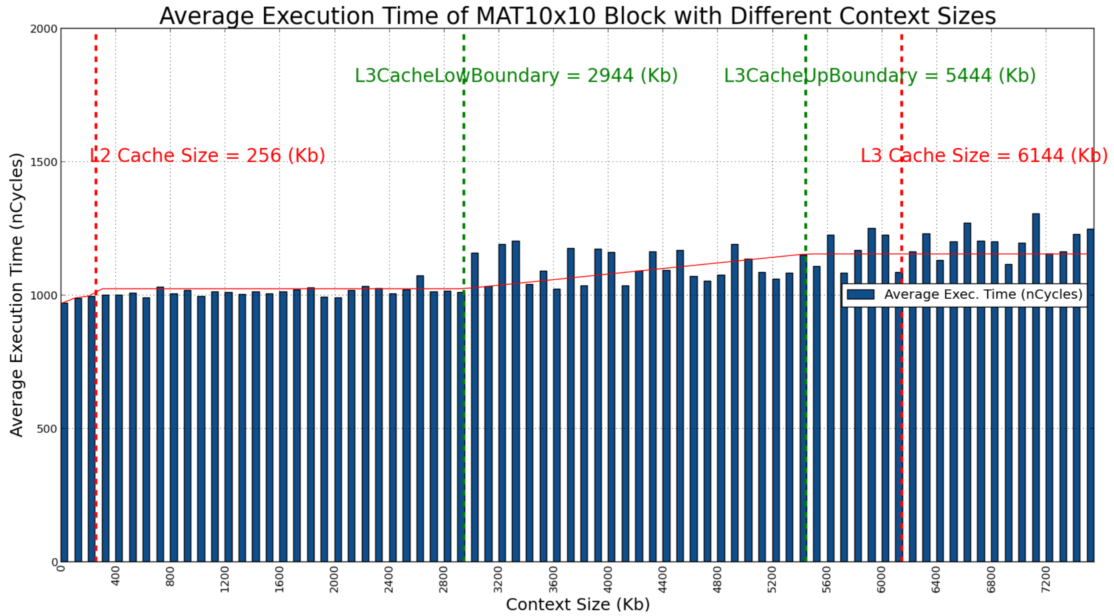

Figure 3.5: Comparison of model predictions with measurements (MAT10x10).

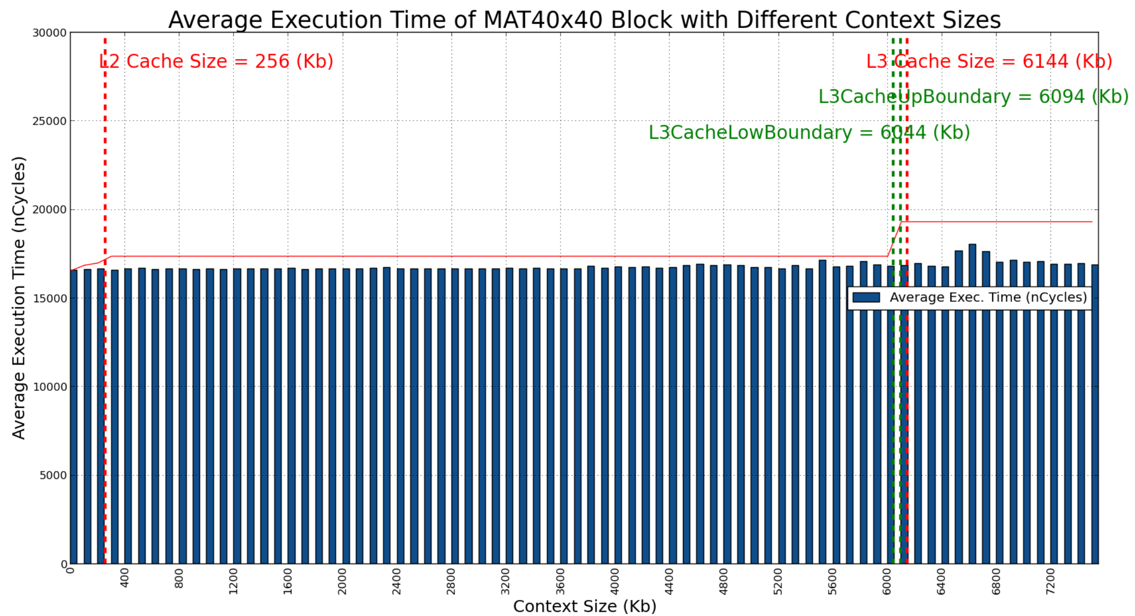

Figure 3.6: Comparison of model predictions with measurements (MAT40x40).

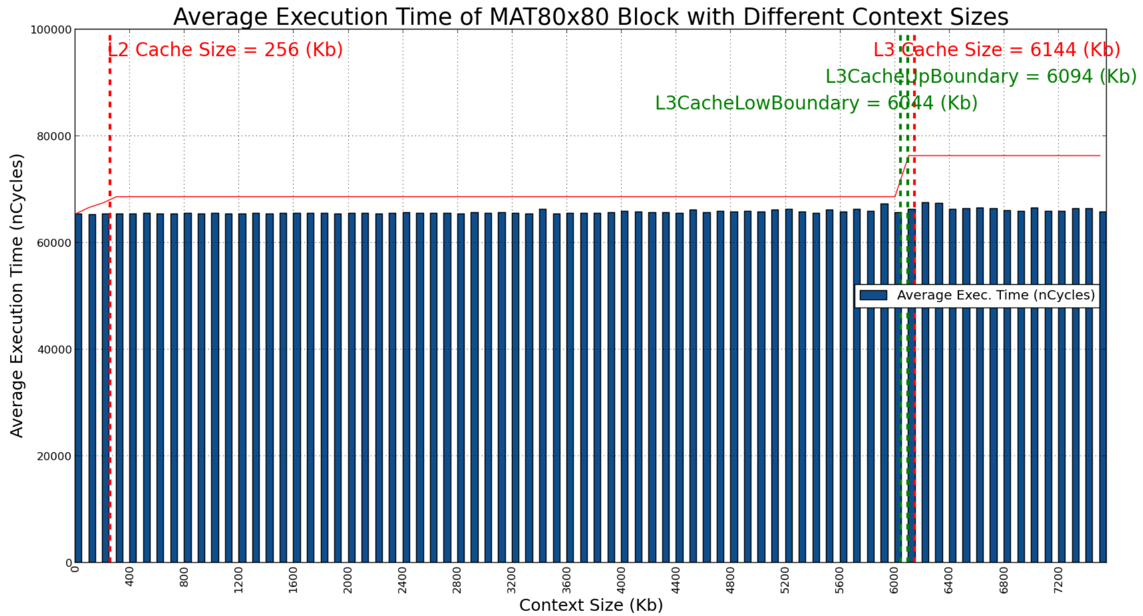

Figure 3.7: Comparison of model predictions with measurements (MAT80x80).

Figure 3.5, 3.6 and 3.7 provide the comparisons of the model predictions (red line) with the average execution times of the task blocks (blue bars) where context size is varied. By analyzing these figures, it is possible to state that the model is able to predict the execution time accurately for the MAT10x10 block. However, the model is intended to overestimate the execution times for MAT40x40 and MAT80x80 blocks. Especially, in the region that maximum number of L3 cache misses are experienced (context size larger than 6144 Kb), overestimation can be observed clearly. Comparison of figure 3.5 and 3.6 shows that prediction of larger task blocks results in higher degree of overestimation. Since as it is illustrated in 2.9, memory latency can be calculated by multiplying number of cache misses with cache miss penalty value, the overestimation of execution time can be caused by one of these parameters. Considering the fact that maximum number of cache misses value is verified by Valgrind, the only parameter that could be overestimated is cache miss penalty. The reason for experiencing smaller cache miss penalty values for the larger blocks can be taking more advantage of the speculative execution circuitries of the execution platform (Hardware prefetcher and out-of-order execution). Consequently, execution platform is able to hide memory latencies more successfully for the larger task blocks.

Figure 3.5 shows that the model has successfully calibrated lower and upper limits of cache boundary region for MAT10x10 block. However, due to inaccurate cache miss penalty values for MAT40x40 and MAT80x80 task blocks, calibration of lower and upper limits couldnt be performed accurately. Since model is developed in a way that estimation error is aimed to be minimized in the calibration phase, it naturally has tried to introduce memory latencies for the L3 cache misses as late as possible. Therefore, model chooses largest possible values for both lower and upper limit parameters while calibrating MAT40x40 and MAT80x80 task blocks.

As a future work for this project, the cache miss penalties can be derived more accurately by performing further investigation on details that may affect this parameter. We believe that as far as this parameter is determined correctly, calibration of the model can be performed without any problem.

3.3 Accuracy of the Predictions

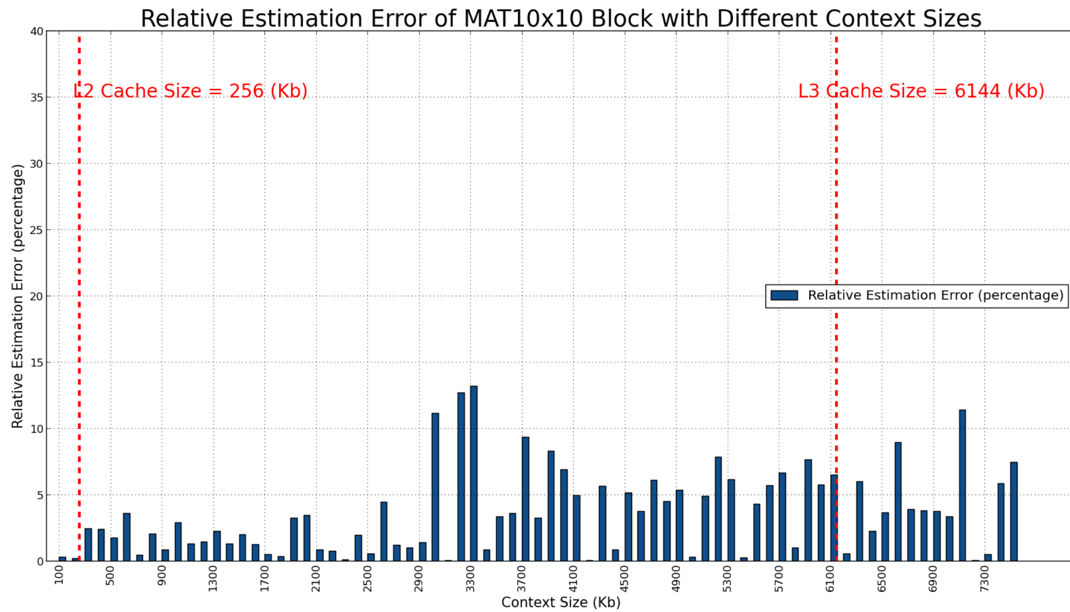

Figure 3.8: Relative estimation errors for predictions of MAT10x10.

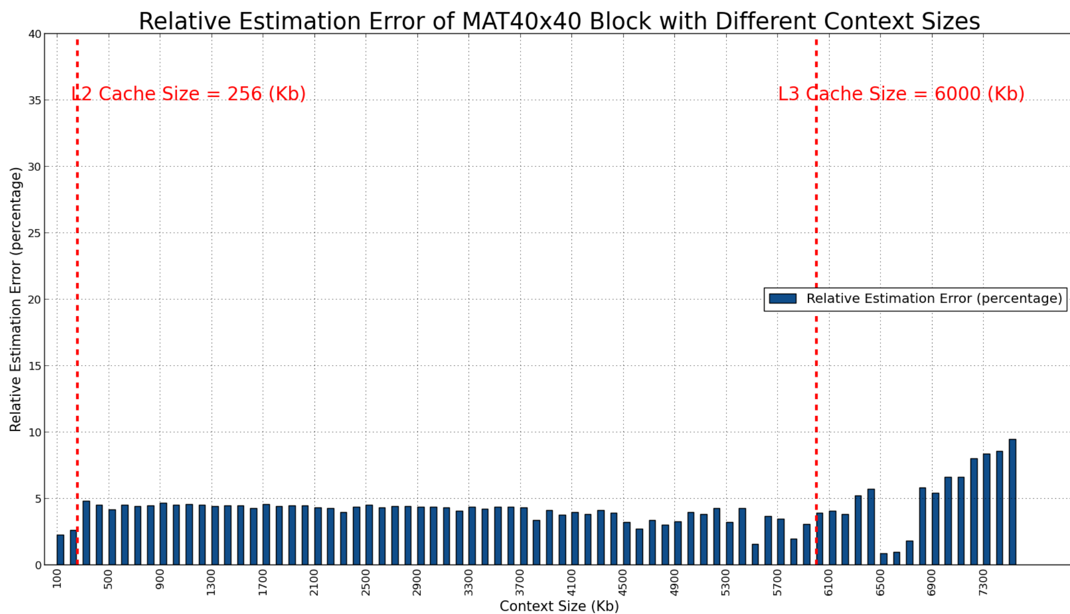

Figure 3.9: Relative estimation errors for predictions of MAT40x40.

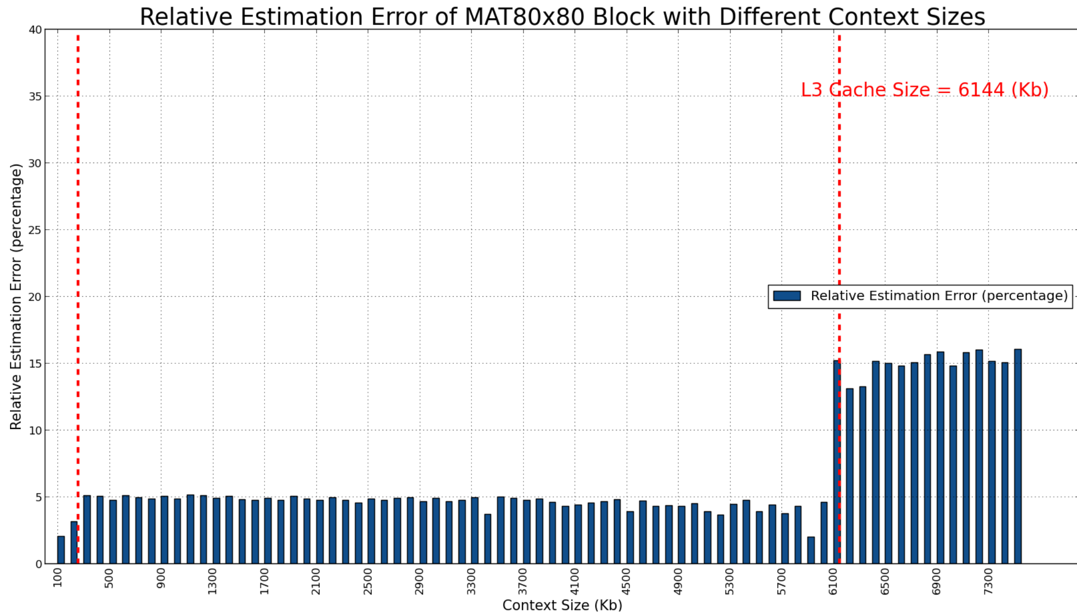

Figure 3.10: Relative estimation errors for predictions of MAT80x80.

Relative estimation errors for MAT10x10, MAT40x40 and MAT80x80 task blocks are given in figure 3.8, 3.9 and 3.10 consecutively. Observation on these figures shows that the accuracy of the predictions is more stable for the larger blocks. Since execution of smaller task blocks takes less time, their execution times are more intended to be affected by any unpredictable behavior of the platform compared to larger blocks. Even if we have assumed that interruptions are completely eliminated by employing RTAI, still it is not an actual real time platform and there may be some other effects on execution time that couldn't be foreseen. Therefore, it is possible to provide more accurate predictions for larger blocks.

Moreover, comparison of these 3 figures implies that, higher relative estimation errors are obtained for the larger task blocks in case context size exceeds L3 cache size. This observation can be explained as follows. Since cache miss penalties couldn't be determined accurately (Smaller cache miss penalties for larger blocks), model overestimates the execution time. Consequently, it less accurate predictions are obtained for larger blocks. In other words, execution platform hides the memory latencies in a more successful way for the larger blocks.

As a result, the model can predict the execution times with less than 20% relative estimation errors for all the investigated scenarios.

3.4 Proof of Feasibility

Since the ultimate goal of this graduation project is to provide a generic model that can predict execution times of any task blocks of ASMLs motion control application which can be mapped given execution platform with at least 80% accuracy, in this subsection, these 2 main concerns (providing generic model and satisfying accuracy constraint) are addressed consecutively.

Providing a generic model introduces two different challenges in the scope of this project namely being able to predict different type of task blocks and modeling any given execution platform. Since throughout the project a limited number of task blocks could be examined, the accuracy results could be provided for only these task blocks including matrix multiplication, filter, and feedback control blocks. However, proposed model abstracts information from task blocks independent of the type of the task block. In other words, the procedure that is presented in section 2.1 can perfectly work for any type of block.

Regarding the support for arbitrary execution platforms, we have tried to develop the model in a way that no platform specific parameters are used. The parameters that are used to represent the execution platform in the model (Table 2.4),abstract several features of the execution platform including the ones that can be specific to a particular execution platform. As mentioned earlier, throughout the project all the experiments are performed on "x86 architecture" based "Intel i7" processor, but in fact "PowerPC" based "Freescale P4080" platforms are employed in ASML's lithography systems to execute the motion control applications. Considering the differences in instruction set architectures of these platforms (CISC vs. RISC machines), one can address the difficulty to represent different features and behaviors of the platforms in the model. However, as described in section 4, the CPI parameter of the model abstracts all the features of the execution platform except for memory latencies based on data reads from L2, L3 and main memory. Moreover, to represent the effect of all modern speculative execution circuitries (which can hide the memory latencies), cache miss penalty parameters (L1, L2 and L3) are used in the model. As long as these parameters can be determined accurately, the model is able to provide satisfactory predictions.

In addition to be able to provide a generic solution, the accuracy constraints of the project should be satisfied. Accuracy of the predictions can be improved by reducing the level of abstraction. In the proposed model, effects of several different features of the execution platform and application are abstracted in few parameters. However, it is possible to perform a further investigation on these parameters and redefine them with several new parameters. Especially, sensitive parameters such as CPI and cache miss penalty can be redefined in case higher accuracy of predictions is desired. As it was mentioned, CPI captures the effects of modern architectural features such as out of order execution, deep pipelining, SMT (Simultaneous multithreading) and superscalar. Moreover, it includes the effect of TLB cache usage. In addition to the CPI, cache miss penalty parameter includes effects of several execution platform features including, cache associativity, architecture of interconnections on network on chip and cache replacement policies. In order to predict the memory latencies these effects can directly be represented in the model by performing further investigations on them.

Chapter 4

Literature Study

A literature survey is conducted by examining related academic work focusing on abstract modeling of applications and platforms. In order to determine the correct level of abstraction, several papers in literature on different levels of abstractions are analyzed in terms of accuracy of their predictions. According to [7], the main trade-off in different levels of abstraction is the fact that a higher level of abstraction requires less effort to model and enables faster analysis. As opposed to its advantages, a high level of abstraction comprises the accuracy of the prediction.

The available academic papers on abstract modeling for applications and platforms can be divided in 2 groups namely, offline (static) analysis based performance estimations and dynamic (run-time) analysis based predictions. In the following parts of this section, these approaches will be discussed consecutively, from the viewpoint of this graduation project.

4.1 Static analysis based methods (Offline)

Performance estimation depending on static analysis can provide worst case execution time (WCET) and therefore can guarantee that no deadline misses will be experienced. This information can be obtained by analyzing all possible execution paths of the application and determining the most critical path. This path can be determined by using ILP (Integer linear programming) techniques. However, these estimations often are significantly larger than the real execution time of the tasks. Since this method often overestimates the execution times, hardware platforms cannot be utilized by employing this approach.

This method can be employed in case hardware utilization is not critical but respecting the hard deadline is. However, one of the main purposes of this project is to utilize the hardware platform as much as possible. Therefore, this approach is not suitable to meet the goals of this graduation project.

4.2 Dynamic information based methods

Dynamic information can be obtained during the execution of the application on the platform. By introducing this information, one can analyze not only cache misses but also the branches in instructions. Therefore, execution time estimations depend on these methods provide higher accuracy.

There are several studies in which different levels of abstractions are employed to model the application and the platform. Following subsections provides analysis of some of these techniques where suitability of these techniques for our project is discussed.

4.2.1 ISS (Low level detailed analysis)

Instruction set simulation based approaches basically execute the application on a virtual platform and provide cycle accurate results. Since these methods not only execute the application but also annotate the information from detailed analysis, performing this simulation requires a larger amount of time than executing the application itself. Obviously, it is not possible to explore the complete design space by simulating all possible mapping combinations.

As far as this graduation project is concerned, although the simulation speed is not a concern, still this method is not feasible in terms of satisfying the providing a generic model. Since the dynamic information is specific to application and the platform, for every different combination instruction set simulations have to be re-performed. This implies that

4.2.2 High abstraction level approaches

There are several academic papers which address execution time estimation based on high level of abstraction. One of these papers is chosen [9] and the suitability of the approach is analyzed in terms of modeling effort and the accuracy of prediction.

According to [9] workload models from an application can be defined by employing a top-down refinement approach. Moreover, a bottom-up composition type hardware platform modeling is realized in SystemC. The abstraction level of models can be modified by either introducing or hiding information about the components. Workload models do not contain timing information. It is left to the platform model to find out how long it takes to process the workload. The bottom layer of platform model consists of basic services that the hardware platform can offer. It has cycle accurate timing information. However the data paths of processing units are not modeled in detail which makes the performance estimation inaccurate. A case study showed that performance estimations can be obtained with a relative estimation error of 20%. [9]

From the viewpoint of this graduation project, even if this approach has advantages such as requiring less modeling effort and adaptability to different applications and platforms, the accuracy of the prediction is still a problem. Since the accuracy constraint of this project is to obtain at least 80% accuracy for all possible cases, it should be guaranteed that predictions with relative estimation error less than 20% are always provided.

4.2.3 Hybrid approach - TLM (Transaction Level Modeling) based approach

One of the common approaches recently developed in academia is to use transaction level modeling (TLM), which makes it possible to hide the computational details of irrelevant processing units [8]. Low level analysis of certain functions/components is performed and the rest of the system is modeled at a higher level of abstraction. This approach provides 3 orders of magnitude faster simulation compared to the ISS approach by compromising %2-3 error in the accuracy of predictions. Timing annotations obtained from compiled binary code is added to the source code (accurate) and this new source code is executed in the platform (faster simulation).

This approach requires a low level analysis of only relevant components and hides detailed analysis of other components. Therefore it is possible to obtain accurate results in a relatively a small amount of time [10], [11].

As a result of our literature study, static analysis based methods are eliminated due to the inaccuracy of the estimations. Moreover, ISS based methods are excluded because the model developed by using these methods cannot provide a generic solution for different types of applications and the platforms. Since our goal is to provide generic solution which can provide accurate predictions, we have decided to employ a method in which level of abstraction is similar as the work presented in either the high level abstraction or in the hybrid approach. The proposed model not only abstracts the application and the platform in a high level but also represents the effect of dynamic information.

4.3 Comparison of our approach with literature

This graduation project proposes a model that abstracts the application and the execution platform in a higher level. However, unlike most of the academic works employs higher abstraction level modeling, it predicts the execution time by considering the dynamic (run-time) information. The academic works that employ run-time information in their models require execution of the application on the platform. However, the dynamic information is introduced to our model in a way that it doesn't require to execute the application on the execution platform. Since we have aimed to provide a generic model by keeping the abstraction level as high as possible, executing the application on the platform and extract dynamic information accordingly was not an option. Therefore, relations between input parameters of a task block and different types of dynamic information are investigated and derived for each different task block type. Consequently, dynamic information is represented in the model as mathematical equations whose input parameters obtained from both the application and the execution platform. According to results presented in section 3, higher level of prediction accuracy is reached compared to the models in the literature that are developed by employing high level of abstraction. Therefore, our approach can be employed in case accurate execution time estimations for the motion control applications are desired without executing the application on the platform.

Chapter 5

Conclusions and Future Work

The main goal of this graduation project is to predict the execution times of the tasks for motion control applications by focusing on modeling. The abstraction level of the model should be determined in such a way that a generic solution for different types of applications and platforms should be provided. On the other hand, the model has to contain sufficient detail to obtain 80-90% prediction accuracy. It should be able to deal with both single-core and multi-core platforms. Therefore, the model should consider the effect of not only overhead due to private resource usage (L1-L2 caches), but also contention in the shared resources (shared memory, off-chip memory and interconnections). It should be possible to change the accuracy of the estimations by tuning the abstraction level of the model.

To this end, this graduation project presents a flexible generic model in which arbitrary tasks of motion control applications and any given execution platform can be represented. The accuracy of the predictions can be tuned by either introducing or hiding some features. The total execution time that model provides, consists of two entities namely pure computation time and memory latency. First, pure computation time is estimated by multiplying the number of instructions with the CPI (Cycles Per Instruction) of the task block that is modeled. Secondly, the prediction of the memory latency is determined by multiplying the number of cache misses with the cache miss penalty value. To estimate the number of cache misses accurately, the effect of the context that the task block is placed in considered. The number of cache misses is predicted as zero, in case total size of the application does not exceed the lower limit of cache boundary region. Moreover, the number of cache misses is provided as the *maximum number of cache misses* if the total application size exceeds the upper limit of the cache boundary region. Since each time the execution platform experiences a cache miss it retrieves a complete cache line, the maximum number of cache misses is provided as the number of cache lines that the application data can fit in. Therefore, the maximum number of cache misses is calculated by dividing the size of the task block by the cache line size.

Since the data size at which the application starts to experience cache misses depends on the application type, the upper and lower limit of the cache boundary region parameters are determined as tunable parameters. For every different type of application, these parameters are re-calibrated. Eventually, the number of cache misses in the cache boundary region is predicted by using linear interpolation between '0' and the maximum number of cache misses. Furthermore, the cache miss penalty values for different levels of cache memory are obtained by using hardware calibrator software.

Experimental results show that the model provides predictions with more than 90% accuracy (average estimation accuracy) for the examined blocks. However, we have experienced that the model overestimates the memory latency for larger task blocks, since the memory latencies are hidden more successfully in these task blocks. Therefore, the accuracy of the predictions can be

improved by examining the variations in cache miss penalty. To this end, the level of abstraction can be reduced by refining the cache miss penalty parameter.

As it is mentioned, the goal of the project is to provide a generic model which can deal with both multi-core and single-core platforms. However, due to time limitations, the model developed for single-core platforms couldn't be evaluated in the multi-core case. In a future project, the model can be adapted for multi-core execution platforms by considering several issues such as contention in shared resources (shared memories, network on chip) and communication overhead between the cores.

Apart from extending the project for multi-core execution platforms, one may consider improving the accuracy of the model. Predictions of the current model can be improved by focusing on the memory latency of the execution platform.

Finally, predictability of the execution platform can be examined by analyzing the distribution graph of execution times.

Bibliography

[1] W. Alberts. "SDS Process Control." Doc ID 102312/07. February, 2007 (ASML Internal Documentation) 3

[2] W. Alberts. "EPS Process Control." Doc ID 116573/30. November, 2010 (ASML Internal Documentation) 3

[3] EPS PG Blocks Control (ASML Internal Documentation) v, 4

[4] EPS PG Blocks General (ASML Internal Documentation)

[5] Schiffelers, R.; Alberts, W.; Voeten, J.P.M. (2012). Model-based specification, analysis and synthesis of servo controllers for lithoscanners. 6th International Workshop on Multi-Paradigm Modeling 2012 Satellite event of the ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems (MPM'12) october 1, 2012, Innsbruck, Austria, Innsbruck, Austria: ACM. 5

[6] CARM 2G Workshop WS0-Multicore scheduling theory WS0_scheduling.pptx January 21, 2013 5

[7] Kyuho Shim; Woojoo Kim; Kwang-Hyun Cho; Byeong Min, "System-level simulation acceleration for architectural performance analysis using hybrid virtual platform system," SoC Design Conference (ISOCC), 2012 International , vol., no., pp.402,404, 4-7 Nov. 2012 30

[8] L. Cai & D. Gajski, [ldquo] Transaction Level Modeling: An Overview,[rdquo] Proc. Int',l Conf. HW/SW Codesign and System Synthesis (CODES-ISSS), pp. 19-24, Oct. 2003. 32

[9] Kreku, J.; Hoppari, M.; Kestila, T.; Yang Qu; Soininen, J.-P.; Tiensyrja, K., "Application - platform performance modeling and evaluation," Specification, Verification and Design Languages, 2008. FDL 2008. Forum on, vol., no., pp.43, 48, 23-25 Sept. 2008 31

[10] Stattelmann, S.; Bringmann, O.; Rosenstiel, W., "Fast and accurate resource conflict simulation for performance analysis of multi-core systems," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, vol., no., pp.1, 6, 14-18 March 2011 32

[11] Stattelmann, S.; Bringmann, O.; Rosenstiel, W., "Fast and accurate source-level simulation of software timing considering complex code optimizations," Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, vol., no., pp.486, 491, 5-9 June 2011 32

[12] http://valgrind.org/docs/manual/manual.html 9, 17

[13] https://www.rtai.org/ 12

Appendix A

Terminology

CARM	The Control Architecture Reference Model is a description to create a well defined layered model of a system where each layer has responsibilities at a specific abstraction level. CARM is meant to be used in a multidisciplinary environment covering the software, electrical and mechanical disciplines.
DoF	A Degree of Freedom is an independent displacement or rotation that indicates the orientation of a system within a three-dimensional space.
HPPC	The High Performance Process Controller is the hardware module onto which the ProcessCtrlWorker is mapped. It is regulated by Bare-board RT OS.
IC	An Integrated Circuit is an electronic circuit fabricated by patterned diffusion of elements into the surface of a thin substrate of semiconductor material.
PGWB	The Process Control Worker Block is the ASML component that contains the worker block derivatives.
SPG	The SetPoint Generator is a worker block that generates subsequent set-points given a profile definition.
UV	Ultra Violet light is the electromagnetic radiation in the range of 10 nm to 400 nm of wavelength.