Software and Network Modeling for Early Design Phases

J. Chr. Braam, J. Rox, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
D-38106 Braunschweig / Germany
{braam|rox|ernst}@ida.ing.tu-bs.de

January 16, 2009

Abstract

There are many use cases in early phases of embedded system design that are interested in reliable estimation of execution times and network loads. This paper presents two new approaches which address these topics. The first one is a formal method which is based on abstract load and execution time data and allows an early characterization and comparison of different SW-to-HW mappings. The other one predicates an self-adapting abstract simulator that provides an estimated upper and lower execution time bound rather than a single estimated value.

1 Introduction

It is very difficult to reliably estimate embedded software and network load in early design phases and even more difficult to estimate embedded system response times. Response times, however, are essential for networked control, such as found in advanced driver assistance systems.

Some of the main challenges of early load and response time estimation are the lack of a complete hardware platform or platform simulation/prototyping model, typically a development environment with a very limited portfolio of supported processors that constrains design space exploration, and, last not least, an incomplete run time environment that doesn't allow to run complete applications. Educated guesses and data of reused software components are typically all that is available to qualify the resulting hardware/software platform. Formal platform models and analysis might point a way out of that dilemma. Today, they are regularly used at several automotive companies to verify the correct timing of an ECU or of a complete network consisting of buses and gateways, i.e. on the right edge of the V-model. The underlying models are an extension of models known from schedulability and network analysis. Since they work with abstract load and response time data, they don't need an executable run time environment or a complete software implementation. Their activation and time functions are compatible to those of Simulink and other tools used in automotive application development. Unfortunately, the application of these formal performance analyses in very early design phases can be difficult, since they need detailed information about certain system parts, e.g. the implemented scheduling policies and the corresponding scheduling parameters, which may not be available yet.

The formal methods, as presented in this paper, are based on abstract load and execution data and are thus applicable before executable hardware or software models are available. The needed data can be estimates derived from previous product generations, similar implementations, or simply engineering

competence allowing for first evaluations of the application and the architecture. Contrary to formal system level performance analyses, it abstracts from schedule or arbitration mechanisms needed to share hardware in time or space. This already allows to tune an architecture for maximum robustness against changes in system execution and communication load, reducing the risk of late and expensive redesigns. It can also support in identifying promising (or very poor) scheduling and mapping decisions. During the design process, these models can be iteratively refined, eventually leading to a verifiable performance model of the final implementation.

A remaining challenge is execution time estimation of a software process. This is needed in many use cases ranging from porting of reused code to a new processor to design space exploration. Due to increasingly complicated processor architectures, this is far from simple and even a pain for processors with an identical instruction set but with a slightly different processor pipeline or memory architecture. There have been many attempts to use statistical and/or self adaptive estimations, but none of them has found wider practical application. The challenge of this problem is to find a suitable abstraction of the execution time model which is able to handle the theoretical huge parameter space of a processor architecture. Using such estimations in the context of formal analysis, however, opens new opportunities. Since schedulability and network analysis work with load bounds rather than exact values, it is now acceptable to use statistics to derive upper and lower bounds for software execution time. The second part of the paper will demonstrate that this approach leads to much more reliable estimations.

The rest of this paper is organized as follows. In Section 2, we give a brief introduction to the compositional system level performance analysis approach underlying SymTA/S. In Section 3, we first introduce the basic concepts for an abstract load model of a system, based on data commonly available in very early design phases. Second, we show how this model can be extended to also consider more sophisticated application behavior like execution time variations. In Section 4, we present a reliable estimation method to obtain the needed execution times of individual SW-functions. Finally, we draw conclusions.

2 Formal System Level Performance Analysis

In past years, *compositional performance analysis* approaches [10, 7, 12] have received increasing attention in the real-time systems community. Compositional performance analyses exhibit great flexibility and scalability for timing and performance analysis of complex distributed embedded real-time systems. Their basic idea is to integrate local performance analysis techniques, e.g. scheduling analysis techniques known from real-time research, into system level analyses. This composition is achieved by connecting the component's inputs and outputs by stream representations of their communication behavior using event models.

While compositional performance analysis can be used before real hardware and software is available, it still needs a lot of detailed information about the system, e.g. the implemented scheduling policies and the corresponding scheduling parameters, which may not be available in very early design stages. Also, in early design stages, the concrete mapping of the application functions to the hardware components may not be decided yet. Therefore, we will only illustrate one of the key concepts used in the compositional

performance analysis, the event streams, since we will reuse it in our load level model.

2.1 Event Streams

The smallest unit modeling performance characteristics at the application level is called a *task*. Timing properties of the arrival of workload, i.e. activating events, at the task inputs is described with an activation model. Instead of considering each activation individually, as simulation does, formal performance analysis abstracts from individual activating events to *event streams*.

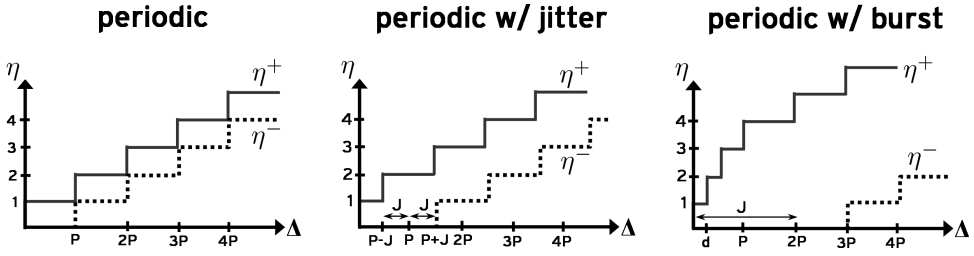

Figure 1: Standard event models

One popular and computationally efficient abstraction for representing event streams are so-called *standard event models* [15], as visualized in Figure 1. Standard event models capture key properties of event streams using three parameters: activation period P, activation jitter J, and minimum distance d. *Periodic* event models have one parameter P stating that each event arrives periodically at exactly every P time units. This simple model can be extended with the notion of jitter, leading to *periodic with jitter* event models that are described by two parameters, namely P and J. Events generally occur periodically, yet they can jitter around their exact position within the jitter interval J. If the jitter value is larger than the period, then two or more events can occur simultaneously, leading to bursts. To describe *bursty* event models, *periodic with jitter* event models can be extended with the parameter d^{min} capturing the minimum distance between the occurrence of any two events.

3 Load Level Modeling

The load level model presented in this section aims at providing predictions about the load generated by the application functions if they were mapped to specific hardware resources. The load level model does not take into account schedule or arbitration mechanisms needed to share hardware in time or space. Neither does it require a concrete SW-to-HW mapping. This makes it applicable before the corresponding decisions are made and it can actually support the designers which arbitration policy or SW-to-HW mapping to choose. The load level model can be seen as the preliminary stage, before conducting a more detailed system level performance analysis, like presented in the previous section.
To help with the decision making, the goal of load level modeling is to provide load profiles for different SW-to-HW mapping alternatives, which describe the load imposed on the resources in a comparable way. Thus, it becomes possible to compare these profiles with load profiles from previous designs, which may

allow an early identification of promising (or very poor) SW-to-HW mappings, based on the experience from older designs.

In the following, we will first present the application scenario, which assumes a Simulink-like model of computation (MOC) of the system to be available. Second, we will introduce the basic concepts of the load level model and explain how they can be used based on the assumed MOC. Afterwards, we will show how he basic load level model can be extended to consider more sophisticated application behavior.

3.1 Application Scenario

Considering control functionality, it is often modeled with CASE tools like Matlab-Simulink. Other CASE tools and their models of computation (MoC) will differ but the principles are the same. Simulink is an extension of Matlab which allows modeling and simulating the desired functionality of complex embedded systems and their environments. It is mainly used in control engineering or signal processing. Value and time continuous or discrete computational models are supported (ref. [Mat07]). As an example consider the block diagram depicted in Figure 2.

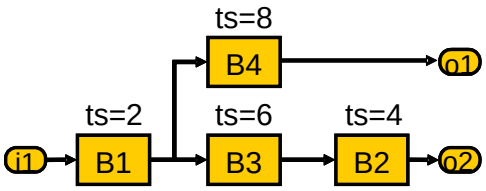

Figure 2: Simulink block diagram

The diagram consists of the blocks B1 - B4 which are interconnected by arrows that represent the communication behavior between the blocks. The blocks are activated or executed only at multiples of a global system clock period, as denoted above each block. This means that the functionality of the block is executed and the corresponding output data is computed. In Simulink the communication between blocks is based on shared variables. During any activation the data at the input of a block is overwritten by the output data of its predecessor block. Reading is possible at any time (none blocking) and deletes no data (non destructive read).

To model the load the different functions generate, two things must be specified for each block:

- The frequency of function activation.
- The processing demand generated by one function execution.

While the frequency of function activation is given by the activation timing of the individual blocks of a Simulink model, the processing demand of one function execution is not included. To obtain the latter, it can usually be estimated by a software engineer or maybe even derived from previous projects, where similar (the same) SW and or HW was used. Another formal estimation method will be presented in Section 4.

3.2 Basic Concepts

The load level model builds upon two basic characteristics of the SW functions. First, the frequency of function activation, which we call the *activation function* and second, the processing demand generated by one function execution, which we call the *software function load (SFL)*.
Formally, an activation function is defined as follows:

Definition 1. *Activation Function:*
The activation function $\eta(\Delta t)$ is a function

$$\eta : \mathbb{R} \rightarrow \mathbb{N} \tag{1}$$

and specifies the maximum number of activations of a SW function in the time interval Δt

As possible types of activation functions we consider the standard event models [15], introduced in Section 2.1. As example, consider a SW function that is activated periodically with period P, but the activation time can jitter around the exact position within the jitter interval J. The corresponding activation function is given by:

$$\eta(\Delta t) = \left\lceil \frac{\Delta t + J}{P} \right\rceil \tag{2}$$

The SFL can be specified in different ways. The simplest one is to specify the SFL by a single value, i.e. the estimated worst case execution time (WCET) of the function. Also more sophisticated SFL descriptions are possible, i.e. cyclo-static load, as will be presented in the next subsection. Clearly, the SFL not only depends on the SW function, but it is also dependable on the HW resource the SW function is executed on. Thus, for each SW function there can be different SFLs for the different HW resources available. For convenience and w.l.o.g., in the following, we consider that only one type of HW resource is available.
We assume the SFL to be given in an informal way, i.e.:

> *One execution of the SW function will take roughly (at most) 30ms.*

From the informal description of the SFL we first determine the aggregate load function, which is defined as follows:

Definition 2. *Aggregate Load Function:*
The aggregate load function $C(n)$ is a function

$$C : \mathbb{N} \rightarrow \mathbb{R} \tag{3}$$

that specifies the maximum time an implementation of a SW function will take to be executed n times.

If the SFL is specified by a single execution time c of the SW function, the corresponding *aggregate load*

function is simply given by:

$$C(n) = n \cdot c \tag{4}$$

From the combination of the *aggregate load function* of a SW function and its *activation function*, we can obtain the processing demand $L(\Delta t)$ the SW function generates over time:

$$L(\Delta t) = C(\eta(\Delta t)) \tag{5}$$

The first characteristic of a load profile, the *average load* $\bar{L}$, can be calculated as:

$$\bar{L} = \lim_{\Delta t \to \infty} \frac{L(\Delta t)}{\Delta t} \tag{6}$$

Assuming a SW function with a periodic or periodic with jitter activation function with period P and a given execution time c, Equation 6 resolves to the commonly known equation:

$$\bar{L} = \frac{c}{p} \tag{7}$$

Note that, the jitter of an activation function has no influence on the average load (for $\Delta t \to \infty$ the parameter J in Equation 2 can be neglected).

The total (average) load generated by several SW functions mapped to the same resource is given by the sum of the corresponding (average) loads of the individual SW functions. In theory, a necessary criteria for a feasible system is that the total average load of each resource must not exceed 1. In praxis however, the upper bound for the allowed *average load* for the resources is usually fixed at lower values to have certain "reserves" in the system.

The *average load* only tells if the processing time provided by the resource is able to catch up with the requested processing or not, but it doesn't predict anything about how long it may take. Hence, to better characterize the load we use the *maximum busy interval*, which is the largest possible time interval in which the resource can be continuously busy, as the second characteristic of a load profile.

Assuming a fully available resource (providing Δt ms of processing time in an interval of size Δt ms), the *maximum busy interval* B is given by:

$$B = \inf\{\Delta t \mid \Delta t > 0,\, L(\Delta t) \leq \Delta t\} \tag{8}$$

As can be seen in the following example, one has to be aware of, that SW functions with different activation functions can impose the same average load on a resource, but have a different *maximum busy interval*.

As an example, consider the block diagram depicted in Figure 2 with the activation periods and execution times as summarized in table 1.

The fourth column of the table contains the *average load* according to Equation 7. From these results it can already be concluded, that mapping B3 and B4 to the same resource is infeasible, since the resulting

Table 1: Execution times

Block	activation period	SFL (WCET)	average load
B1	20ms	5ms	0.25
B2	40ms	5ms	0.125
B3	60ms	30ms	0.5
B4	80ms	50ms	0.625

total load would be greater than 1.

If we consider mapping the SW functions represented by B2 and B3 on the same resource, Figure 3 illustrates the resulting load, which is the sum of the loads of the two blocks, obtained using Equation 5.

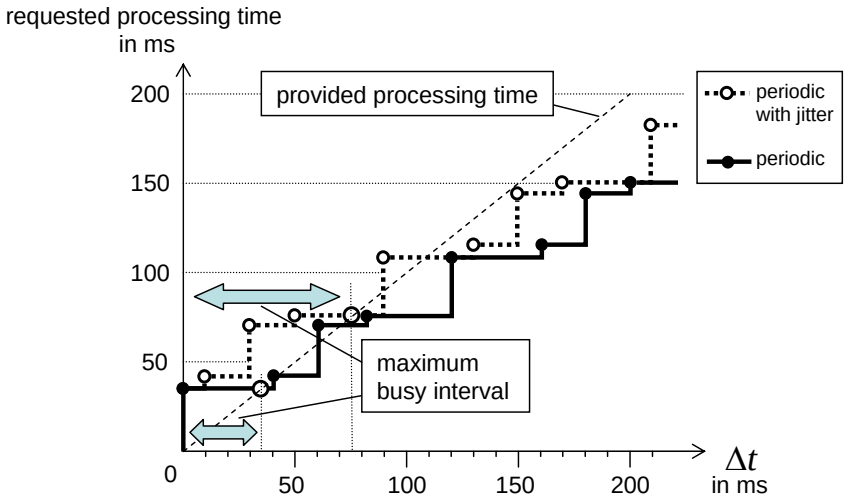

Figure 3: Total load of B2 and B3

In the figure, the load for two different activation functions are shown. The solid line shows the load for the strictly periodic activation of both SW functions with a period of 40 ms for B2 and 60 ms for B3. The discontinuous line shows the load for the case, where both SW functions have an additional activation jitter of 30 ms. The dotted straight line with slope 1 illustrates the maximum processing time a fully available resource could provide (Δt ms of processing time in an interval of size Δt ms).

The first intersection point of the provided processing time and the required processing time marks the end of the *maximum busy interval* (see Equation 8). Although, for both activation scenarios we have the same *average load*, the size of the *maximum busy interval* more than doubles if we add jitter to the activation function. Just comparing the two different load situations on the basis on the *average load* wouldn't show any difference, but also considering the *maximum busy interval* the distinction becomes clear.

Note, that the above load model doesn't make a distinction between communication time or computation time. Hence, by specifying the required communication time (equivalent to the SFL) of a certain message together with the release frequency of this message (equivalent to the activation function), load profiles for networks or buses can also be determined in the same way.

So far, the SFL of a SW function is only characterized by a single execution time. If this is the WCET, it

is guaranteed to deliver conservative results, but these results can easily become very pessimistic. Hence, in the following subsection we will present how we can also consider more detailed description of the SW function behavior.

3.3 Considering Execution Time Variations

If more detail about the internal behavior of the SW function is available, it can also be specified more precisely. For example, the SFL of the Block B3 could be characterized to have a cyclo-static execution behavior, described as follows:

> *The SW process cycles through a sequence of five steps which will take roughly 5ms, 30ms, 15ms, 10ms, 20ms.*

Another possibility to describe the SFL is a scenario dependent behavior. E.g. the SFL of block B4 could be specified by:

> *The SW process uses 50ms or 20ms, depending on whether it must correct an error or not. Out of 5 executions, at most 1 error must be corrected.*

While the WCETs of the two functions are still 30ms and 50ms, the load they impose on a resource is obviously smaller (the average load, as well as the transient load).
To calculate the *average load* of a SW function with a cyclo-static execution behavior, we first determine the total requested processing time for one complete cycle C_{cycle}, as follows:

$$C_{cycle} = \sum_{i=1}^{|E|} e_i \tag{9}$$

Where $e_i \in E$ is the execution demand of the i-th step of the SW function.
Hence, for the block B3 with $E = \{5ms, 30ms, 15ms, 10ms, 20ms\}$ we obtain $C_{cycle} = 80\,ms$.
Second, we determine the activation frequency of a complete cycle P_{cycle} by multiplying the activation period P of the SW function by the number of elements of E:

$$P_{cycle} = P \cdot |E| \tag{10}$$

To calculate the *average load*, we can now use Equation 7 with T_{cycle} and P_{cycle} instead of c and p. So, for the block B3, we obtain a P_{cycle} of 300 ms which results in an average load of $\frac{80}{300} = 0.267$ for the SW function modeled by B3.
To be able to use Equation 5 on SW functions with a cyclo periodic function load, we must determine the corresponding *aggregate load function*. We do this in two steps. We start by determining the load of n consecutive executions, with $0 \leq n < |E|$. We can do this by looking at each possible interval of size n and taking the maximum. E.g. for 2 consecutive executions of the block B3, we find a maximum requested processing time of 45 ms, as illustrated in Figure 4.

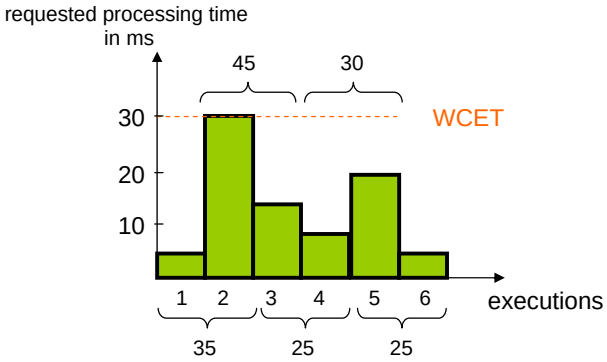

Figure 4: Possible load of 2 consecutive executions of Block B3

Formally, we can express this calculation of the maximum sum for a given interval size by the function

$$F(n,E) = \begin{cases} 0 & : n = 0 \\ \max\limits_{j \in [0,|E|-1]} \sum\limits_{i=j}^{j+n-1} e_{(i \bmod |E|)+1} & : 0 < n < |E| \end{cases} \tag{11}$$

where $e_i \in E$ is the execution demand of the i-th step of the SW function.

Now we can calculate the requested processing time for $n \geq 0$ consecutive executions of such a cyclo-static SW function by determining the load imposed by the number of complete cycles and using Equation 12 for the remaining executions. Hence, the maximum load generated by n consecutive executions of a SW function with a cyclo-static function load specified by $E = \{e_1, .., e_k\}$ is given by:

$$C(n) = \left\lfloor \frac{n}{|E|} \right\rfloor \cdot T_{cycle} + F(n \bmod |E|, E) \tag{12}$$

According to the specification of the SFL of block B4, the most processing time is requested if every 5 executions an error occurs. Obviously, this behavior can again be interpreted as a cyclo-static behavior with $E = \{50, 20, 20, 20, 20\}$. Hence, the load imposed by n consecutive executions of block B4 can also be expressed using Equation 12.

Figures 5 a) and b) show the load of consecutive executions of the block B3 and B4 just assuming the WCET (using Equation 4) compared to the more detailed cycle periodic function load for B3 and B4(using Equation 12).

Calculating the *average load* for B3 and B4 we obtain 0.267 for B3 and 0.325 for B4. Since the sum of both *average loads* is smaller than 1 now, mapping the SW functions modeled by B3 and B4 to the same resource isn't outright infeasible anymore.

Assuming that B3 and B4 are mapped to the same resource, we can obtain the resulting load profile by using Equation 12 and Equation 5 for both SW function and adding up the results. The outcome is illustrated in Figure 6.

As one can see, the *maximum busy interval* has a size of 110 ms, if we consider the execution time variations. Only considering the WCET of the functions overloads the resource, which in the graphical representation means, that the curve representing the requested processing time and the curve of the

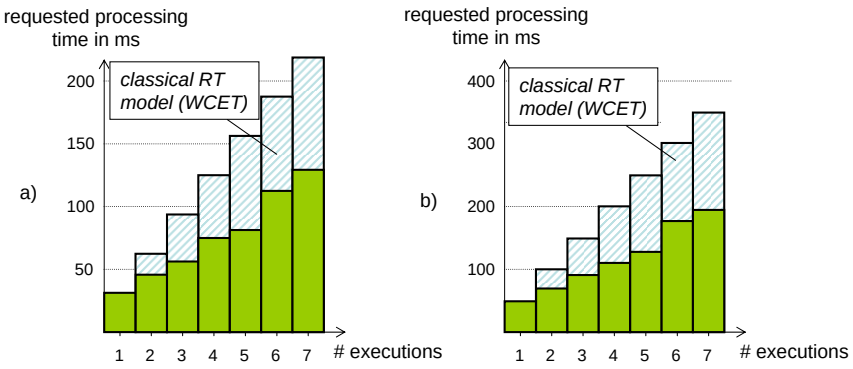

Figure 5: a) Requested processing time of consecutive executions of B3, b) Requested processing time of consecutive executions of B4

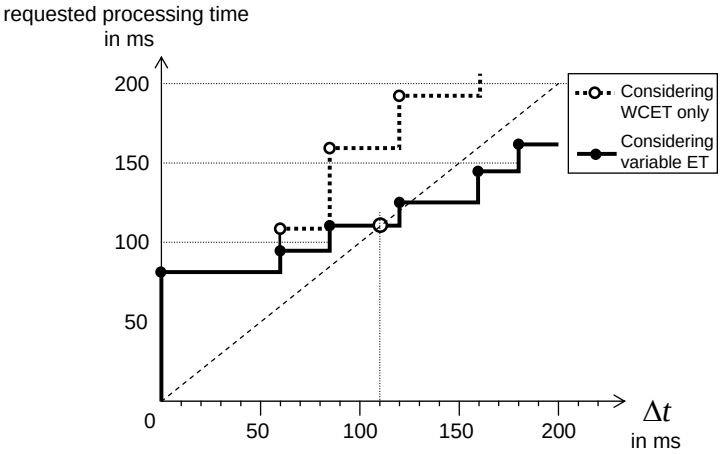

Figure 6: Requested processing time vs provided processing time

provided processing time never intersect.

As can be seen from the above example, the determination of the load profile for a function or a resource can also be done for different SW function behaviors, given by an informal description. The only requirement for this description is, that it allows to derive the mathematical function describing the *aggregate function load*. Obviously, the more accurate the SFL of a SW function is, with respect to its real execution demand, the more accurate the resulting load profiles are.

To obtain accurate execution data of the single SW functions, in the following section, we will present a reliable estimation method, that can deliver the execution times of a SW function for different execution scenarios. This estimation method not only complements the here presented load profiling, but the resulting estimates are also an important input for a formal system level performance analysis like presented in Section 2.

4 Single Process Execution Time Estimation

The load level modeling presented in Section 3 requires the execution times of SW functions. In early design phases this is often based on educated guesses because no more information is available. But,

if the SW functionality can be described by an implementation language (e.g., "C"), a more accurate estimation is possible. This is possible in many cases. First, automotive projects are often evolutionary and therefore much code is reused. And secondly, common CASE tools like Matlab-Simulink offers code generators for generic "C" code. An intuitive solution would be measuring with real HW. But, this is no way out because the setup and usage of such a test framework is very complex and requires excellent HW knowledge. The situation will be hampered if there are many SW developers which need execution times for different SW functions and HW (e.g., architecture exploration).
The previous mentioned use case is assumed for our self-adapting estimation method. This approach provides reliable execution time intervals with upper and lower estimation bounds, rather than single estimated execution time values. Here, reliable means that major individual estimation errors are not to be expected. Due to the methodology principle, the results are not conservative.
In later design steps, when more information regarding the system configuration (e.g., SW mapping, communication topology, etc.) is available, the results can be used easily by the performance system analysis (Section 2). The timing intervals are compatible to the input requirements of the analysis tool.
This chapter is structured as follows. We begin with an overview of estimation methods in Section 4.1. Details of our estimation method are described in Section 4.2. Next, we explain our experiments and discuss the results in Section 4.3. The chapter ends with an executive summary in Section 4.4.

4.1 Related Work For Estimation Methods

In recent years, other approaches that target the prediction of SW execution time have been developed. Currently, cycle accurate simulators seem to be the best choice for measuring the execution time of single processes. Cycle accurate means that the states of the HW model and of the real HW are identical on clock boundaries [3]. Commercial simulators supporting such accuracy and featuring high simulation speed are currently available, e.g. Virtual Platform Designer (http://www.coware.com), RealView Maxsim (http://www.arm.com) or METeor (http://www.vastsystems.com). The limitation of this approach is that only a subset of available real HW components is available as cycle accurate models, especially in the automotive sector. Due to volume, architectural complexity and number of applications, the focus is on models for the consumer and mobile communication industries. Although the model creation is assisted by model generation features of the tool sets mentioned above this step is still very time consuming and requires excellent HW knowledge.
Another technique is based on a pseudo machine with a virtual instruction set, which is annotated with timing information. Detailed information can be found in [13], [9] and [4]. SW (C code) is transformed in two ways. Either the target SW (C code) is compiled with the target compiler, or with a source annotator or "virtual compiler", which replaces the target compiler. In both cases, the resulting object code is transformed into virtual instructions that will later be simulated. Each instruction has a fixed timing and the delays of all executed instructions are accumulated. Suitable timing annotations are found via a training phase. A set of benchmarks runs on the virtual HW and on the real HW. The execution time on the real HW is measured and the executed virtual instructions are counted. The measured time and the number of virtual instructions of each benchmark are correlated in a linear regression analysis.

Based on the least square metrics, the execution time of the different virtual instructions is computed. The limitation of this approach is that the abstract model of the real HW is very simple. Therefore, the inaccuracy will be high in the case of complex pipelined processors (for more details regarding HW principles see [11]).

A similar approach can be found in [6]. The author presents a comprehensive timing estimation process in general and propose a new timing model, clarified with a simple example. The special non-linear timing model estimation technique is called lazy learning. During a training phase, the values of suitable SW parameters (so-called signature; e.g. frequency of executed instructions) is collected and stored with the corresponding measured execution time in a database. During the estimation phase, the signature values of the application SW are generated. Then, these values are compared with the recorded values of the benchmarks. Based on a distance metric, the most similar benchmarks (or rather their signature values) are selected and weighted. Next a regression analysis function has to be chosen for this local subset. The example uses a linear regression analysis based on least square error. Finally, the regression analysis function is applied to the signature value of the SW to be estimated. The limitation of this approach is the need for suitable signatures. They must be able to characterize the timing influences of HW and SW precisely. There are no known systematic solutions for this problem.

Another approach extends the idea of non-linear models. In [14] a solution is described that is based on neural networks (NN) and a classification of the SW. For each SW class whose classification is based on a simple metric a NN is trained with benchmarks or rather its signature values and the corresponding real execution times (measurement). The drawback of this approach is that one needs a suitable signature and an adequate neuronal network type/topology.

The main problems, however, are creating a suitable model, the large training set and the unknown result quality.

4.2 Execution Time Estimation With Intervals

The foundation of our method is a customizable processor simulator (see Figure 7), which focuses on timing and not on functionality. It uses traces of executed instructions, which are created by a fast instruction set simulator (ISS). This kind of simulator was selected because it is easily available for most embedded architectures. ISSs are typically included in the software development environment (SDE) for SW engineers to verify the functional correctness of the SW. This kind of estimation enables the consideration of timing effects caused by the target instruction set architecture and the corresponding compiler or its optimizer.

The simulator consists of a target-dependent module and a target-independent module. The first one is similar to a parser. It transforms the executed instruction (assembly) trace into a generic format. Presently, it consists of: instruction-ID, read registers, written registers, data memory address and instruction memory address. This parser has to be rewritten for each new instruction set architecture. The second module of the simulator reflects the pipeline behavior, which includes many micro-architectural features. They are implemented in a target independent manner. So they can be reused and the adaptation to other processors can be done easily and quickly.

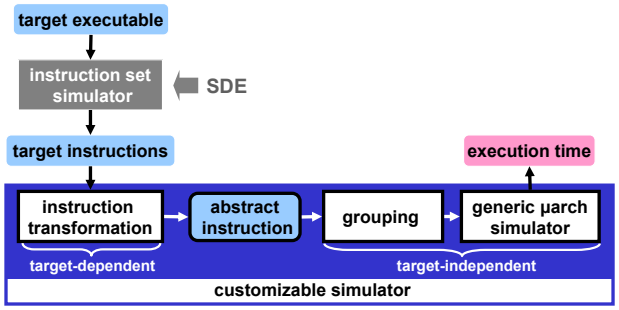

Figure 7: Simulator internals

The length, structure and features of the pipeline are configurable. At the moment, we are supporting scalar in-order execution pipelines with a wide range of structures (e.g., pipelines consisting of two sub-pipelines for arithmetic and load/store operations). Branch behavior and data hazards can be adjusted. Additionally, the duration time of each ID can be specified for each stage of the pipeline. To reduce the configuration effort, it is possible to group IDs that show the same timing behavior in all pipeline stages. This step is optional and requires detailed HW knowledge.

Because it is not possible to cover all existing pipeline features with one simulator, we provide the simulator with an interval-based timing model (see Figure 8).

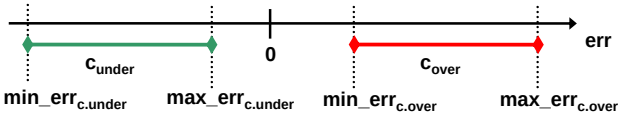

Figure 8: Interval based timing model

The intention behind this idea is that effects, which cannot be modeled precisely with the simulator, will be over- and underestimated to increase the reliability of the estimation. This parameter adaptation is done during a training phase. The configuration space of the simulator is explored with benchmarks to find two suitable configurations. One configuration leads to execution times for all benchmarks that fall below the measured real execution time (green/left line; Figure 8). The other configuration overestimates the real execution time for any benchmark (red/right line; Figure 8). If new SW is to be estimated, these two configurations are used to predict the execution time. The result is a min/max interval of the predicted execution time, which should include the real execution time. This result is compatible with the system analysis. Of course, these bounds are not conservative. If the benchmarks do not stress HW characteristics that are significantly used by new SW, the real execution time will lie outside the estimated interval. But, due to the methodology, the probability of this case is minimized and this is the major objective of early execution time estimation in early design phases.

After a brief introduction of the simulator principles, we will delve into details of the activities that are necessary for time estimation. As previously indicated, the proposed method is divided into the training and a prediction phase. We begin by explaining the training phase.

4.2.1 Training

Before the adaptation of the parameters can start, a set of benchmarks has to be chosen, which is able to stress as many timing effects of the processor as possible. Using SW snippets that focus on different mathematical operations, data types, data values and control flow structures are a good initial point. But, finding a good set of benchmarks is a non trivial step and requires a separate analysis, which is not the focus of this paper.

The execution times of theses benchmarks are measured with real target HW or a cycle accurate simulator. Additionally, the executed instruction traces are generated with an ISS and are recorded. Both parts serve as reference.

The next step is to create the initial simulator configuration. Its configuration is derived from datasheets of the selected processor or similar technical documents. Distinct parameters are adjusted and fixed, e.g. pipeline length/structure and known ID timings. Each parameter whose configuration is unknown is annotated with a set of possible solutions or values. Because the search or parameter space increases exponentially, the size of the set can be limited by basic architecture knowledge (e.g., min/max values for the timing of undetermined IDs). This set shall be large enough to over- and underestimate the real value.

With this setup the self-adaptation procedure can start. It is shown in Figure 9.

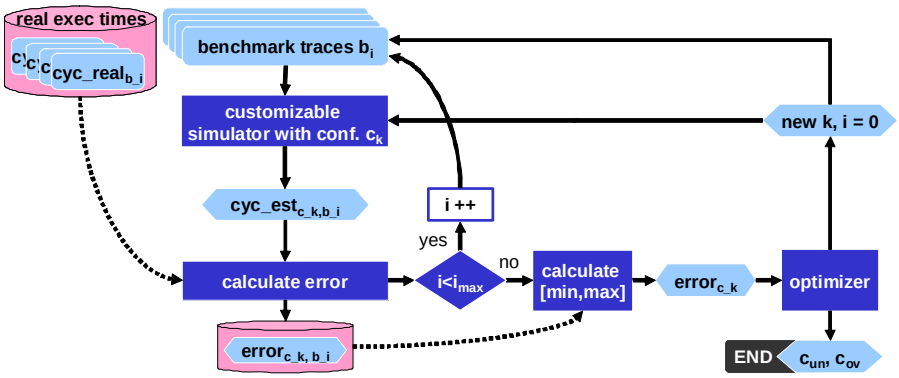

Figure 9: Training procedure

Starting-point is the optimizer module. It creates an initial simulator configuration c_0 consisting of the unknown parameters and starts the simulation with the first benchmark b_0. An execution time is estimated and the error related to the real execution time is calculated (equation 13).

$$err_{c_k,b_i} = \frac{cyc\_est_{c_k,b_i} - cyc\_real_{c_k,b_i}}{cyc\_real_{c_k,b_i}} \tag{13}$$

The result err_{c_0,b_0} is saved and the next benchmark of the set is simulated with the same simulator configuration. After simulating all benchmarks, the minimum and maximum error of all results are

computed (equations 14-16) and the result err_{c_0} is transferred to the optimizer.

$$min\_err_{c_k} = \min_{\forall b_i}(err_{c_k,b_i}) \quad (14)$$

$$max\_err_{c_k} = \max_{\forall b_i}(err_{c_k,b_i}) \quad (15)$$

$$err_{c_k} = [min\_err_{c_k}, max\_err_{c_k}] \quad (16)$$

Based on an efficient search algorithm the optimizer creates a new simulator configuration and starts the simulation with the first benchmark of the set. This loop iterates as long as the predefined termination condition of the optimizer is fulfilled. During this runs, the optimizer keeps the best results in mind. If the optimizer terminates, it will show the best configuration for over- (c_{ov}) and underestimation (c_{un}). To clarify the result and the principle the described optimization criterion is formulated mathematically as follows:

$$C_{over} = \{c_k \,|\, min\_err_{c_k} \geq 0\} \quad (17)$$

$$C_{under} = \{c_k \,|\, max\_err_{c_k} \leq 0\} \quad (18)$$

$$c_{ov} = \arg \min_{\forall c_k \in C_{over}} (max\_err_{c_k}) \quad (19)$$

$$c_{un} = \arg \max_{\forall c_k \in C_{under}} (min\_err_{c_k}) \quad (20)$$

Solving the previous mentioned optimization problems (see equations 19 and 20) requires an efficient search algorithm because the problem or parameter space tends to be very large. Therefore, we choose a genetic algorithm (GA), which is coupled with our simulator by using the PISA protocol [5]. This kind of heuristic search algorithm has many advantages. First, no special knowledge of the problem space is necessary. Secondly, the adaptations to a special problem require little effort. Last but not least, GAs are able to find global optima and does not get caught by local optima (e.g., Hill Climber).

The development of GAs is inspired by natural evolution. The fittest individuals of a population will survive. The principle is briefly described in the following. For more detailed information regarding GAs see [8].

At the beginning, an initial population is created randomly. It consists of individuals with different genotypes (simulator configurations c_k). These candidates are evaluated as described before (calculating the min/max interval for the benchmark set). The result of the evaluation is called fitness or objective values. After these preliminaries the algorithm enters the main loop. First, parents are selected from the population randomly. Then, their genotypes or parameter settings are recombined and mutated based on a stochastic process. The offspring are subjected to a survivor selection based on their fitness values. For this selection step we use the problem independent tool SPEA2 [16]. Now, a new population is generated and the loop starts again except the termination condition is fulfilled. The termination condition is realized by limiting the number of iterations.

Because it is possible that parameter variation of the simulator does not affect the execution time of all benchmarks of the set, the same error interval for different configurations can occur. Therefore multiple solutions for c_{ov} or c_{un} are possible. An additional criterion is necessary to find unique solutions. A

suitable metric is the Euclidean norm, which is applied to all estimation errors of the benchmark set for each valid configuration. The configuration with the minimal norm is chosen (equation 21).

$$\tilde{c}_{ov|un} = \arg \min_{\forall c_{ov_n|un_m}} \sqrt{\sum_{\forall b_i} \left(err_{c_{ov_n|un_m}, b_i}\right)^2} \tag{21}$$

4.2.2 Estimation

During this phase, the execution time of a new SW function is predicted with the simulator shown in Figure 7 and the results of the training.

At the beginning, the trace of executed instructions is created by the ISS that is used by the SW developer or system designer in this phase. If this ISS differs from the ISS of the training phase, the output format and content has to be reviewed. The parser or the instruction transformation module has to be adapted, if applicable. The abstract instructions are grouped in the same manner as during the training. The resulting trace is processed by the simulator two times. The first run is based on the configuration c_{ov} and the other one on c_{un} (see equations 19 and 20).

The results are the predicted min/max number of cycles, which have to be multiplied with the clock cycle of the HW to get the min/max execution times. Equations 22 and 23 clarifies this.

$$t\_est_{max} = cyc\_est_{SW,\tilde{c}_{ov}} \cdot t\_cyc\_real \tag{22}$$

$$t\_est_{min} = cyc\_est_{SW,\tilde{c}_{un}} \cdot t\_cyc\_real \tag{23}$$

4.3 Experiments

The presented methodology shall be clarified with an experiment, which demonstrates the whole methodology including all steps of the training and the estimation phase described in Section 4.2.

The HW environment is based on the ARM9TDMI core, which is a RISC processor with a five stage pipeline (for more details see [2]). During the training and the prediction phase we use the ARM SDE (so called ADS 1.2) for compiling, trace creation and measurement (cycle accurate).

Due to simplification but without any loss of generality some restrictions are imposed: (1) The core uses a simplified memory model (zero wait state) and has no floating-point coprocessor. (2) As the ARM instruction set architecture is quite complex, only a common subset of instructions is supported. E.g., no thumb, no co-processor specific and no Current Program Status Register related instructions are supported at the moment.

4.3.1 Training

First, we start building the parser. The required information for the transformation and the grouping module can be found in [1] and [2]. Additionally, this documentation is used to create an initial simulator configuration (pipeline structure, known group ID timings, data hazards and branch strategy).

Considering the limitations, 29 instruction groups are identified, which need a suitable timing for each pipeline stage. It is assumed that a total of 9 group ID timings of different pipeline stages are not known

and have to be estimated. Since the timing of some instructions cannot be reflected precisely by the simulator (e.g., instructions with a operand data size dependent timing) and the data hazard modeling is not perfect, over- and underestimation of the execution timing are valuable. Although the timing of the unknown group ID timings are restricted by basic technical knowledge, the search space contains ca. 7.8 million combinations.

For the training phase 18 benchmarks, which provide standard functionalities, e.g. sort algorithms, FIR filter and matrix multiplication are selected. Their trace is created by the ISS and processed by the simulator. The genetic algorithm controls the heuristic search. Based on equations 13-21, adequate results ($\tilde{c}_{ov}$ and $\tilde{c}_{un}$) are calculated. The termination condition of the GA is realized by restricting the maximal number of iterations (generations) to 200. Each generation creates 10 new offspring (parameter configurations) based on a stochastic process. Therefore, up to 2000 different parameter configurations are analyzed.

4.3.2 Estimation

The estimation is performed with 12 automotive EEMBC benchmarks. These benchmarks are not part of the training set. Each benchmark is simulated with the over- and underestimation configuration ($\tilde{c}_{ov}$ and $\tilde{c}_{un}$) of the training. The calculated minimal and maximal error for each benchmark is visualized in figure 10. The objective of the methodology is fulfilled that all benchmarks over- and underestimate the real execution time. The error ranges from -1% to 5%.

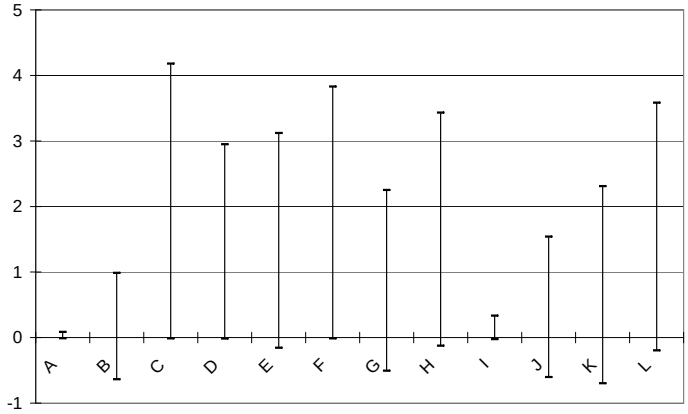

Figure 10: Execution time estimation error (in %)

4.4 Executive Summary

This chapter focuses on the estimation of SW execution time in early design phases, which is essential for a reliable analysis of the system timing behavior. Existing approaches are based on very precise (cycle accurate) processor models for simulators or abstract models, which are adjusted to the individual processor by statistical or self-learning methods. But, the approaches achieve only suboptimal trade-offs between estimation accuracy, flexibility (adapting to different processors) and model creation effort.

Our new approach overcomes limitations of different approaches and results in a better compromise of

the mentioned criteria. It is based on a qualification process that is executed once per processor. That qualification process is expected to be outsourced to a service provider. Its foundation is a self-adapting, parametrizable simulator combined with an interval-based estimation technique. At the moment, the simulator is supporting basic architectural features. A priori known information found in datasheets or similar technical documents can be transferred easily to the simulator configuration. Unknown timing parameters are calculated during a self-adaptation process that is based on a genetic algorithm to efficiently handle huge parameter spaces. Because the simulator parameters cannot represent all HW timing effects precisely, two parameter configurations are used for the estimation. One overestimates and the other one underestimates the real execution time. First experimental results show that this approach is promising. Of course, supporting a wide field of modern processors requires more micro architectural features, which have to be implemented. But, the principle is still the same.

5 Conclusion

While formal platform models and analyses are used today to verify the correct timing of an ECU or a network in the later stages of the design process, it is problematic to apply them already in the early phases of the design. The major difficulty is that the used formal approaches need more specific information about the system than is yet available early on in the design process. Here, we presented a formal approach that based on a model of computation of the application and estimates of the execution behavior of the different SW functions, delivers individual load profiles of the SW functions, which can support in finding promising SW-to-HW mappings. We also presented a reliable method to obtain the needed execution behavior of SW functions on a specific HW platform in form of estimated execution times; given the SW functionality can be described by a implementation language (e.g., "C"). This estimation method not only complements the presented load profiling, but the resulting estimates are also an important input for a formal system level performance analysis in later stages of the design to verify the correct timing behavior.

References

[1] ARM Limited. *ARM Architecture Reference Manual*, 2000. ARM DDI 0100E.

[2] ARM Limited. *ARM9TDMI Technical Reference Manual*, 2000. ARM DDI 0180A.

[3] B. Bailey, G. Martin, and T. Anderson. *Taxonomie for the Development and Verification of Digital Systems*. Springer, 2005.

[4] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. A. Harcourt, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES*, pages 82–86, 2000.

[5] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA - A Platform and Programming Language Independent Interface for Search Algorithms. In *Proc. of the Conference on Evolutionary Multi-Criterion Optimization (EMO)*, Faro, Protugal, April 2003.

[6] G. Bontempi and W. Kruijtzer. The use of intelligent data analysis techniques for system-level design: a software estimation example. *Soft Comput.*, 8(7):477–490, 2004.

[7] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 190–195, 2003.

[8] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2nd edition, 2007.

[9] P. Giusto, G. Martin, and E. A. Harcourt. Reliable estimation of execution time of embedded software. In *DATE*, pages 580–589, 2001.

[10] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The SymTA/S Approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.

[12] T. Henzinger and S. Matic. An Interface Algebra for Real-Time Components. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2006.

[13] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 167. IEEE Computer Society, 2000.

[14] M. Oyamada, F. Zschornack, and F. R. Wagner. Accurate software performance estimation using domain classification and neural networks. In *SBCCI*, pages 175–180, 2004.

[15] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, 2004.

[16] E. Zitzler, M. Laumanns, and L. Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Department of Electrical Engineering, Swiss Federal Institute of Technology (ETH) Zurich, 2001.