SINGLE PROCESS EXECUTION TIME ESTIMATION IN EARLY DESIGN PHASES FOR SW DOMINATED EMBEDDED SYSTEMS

Joern Christian Braam[1], Rolf Ernst[1]
[1] *Technical University of Braunschweig, Institute of Computer and Communication Network Engineering (IDA)*
Hans-Sommer-Str. 66, D-38106 Brunswick
Phone: (+49) 531 391 – 3734, Fax: (+49) 531 391 – 3750, Email: {braam | ernst}@ida.ing.tu-bs.de

Abstract: An important challenge in automotive system design is to estimate the execution time of SW for different HW architectures in early phases (architecture exploration). We present a methodology which enables single process execution time estimation in early design phases. System analysis tools can use the estimated time to calculate the system timing behavior. The methodology minimizes the usage of real HW or simulators and emphasizes portability to different processor types. The approach is based on instruction traces, annotations by a generic simulator and regression analysis techniques. Therefore instruction set architecture, micro architecture and compiler effects (code optimization) are considered. We show some successful experimental results and discuss limitations of the approach.
Keywords: estimation, prediction, execution time, embedded system, software, hardware.

1. INTRODUCTION

Current automotive embedded systems are very complex due to large functionality which is distributed on many networked electronic control units (ECU). The ECUs are often based on standard HW components. Customized HW components, e.g. ASICs, are used only in parts. The development of such systems is distributed among several suppliers. Due to cost reasons a rough system design is fixed very early in the design process. This allows maximizing simultaneous activity and therefore minimizing the development time. This means that mapping of SW functionality to the different ECUs must be finalized early. The consequence of freezing the system very early in the development process is that the system decision requires very careful considerations and planning to avoid cost intensive redesign cycles because of system errors that are detected too late.

Many functional and non functional issues have to be considered. One of them is the timing behavior of SW dominated embedded systems whose importance increases rapidly due to the rising complexity of automotive systems, as reflected in the AUTOSAR software standard (AUTOSAR home page). Classical methods become more and more inapplicable to predict the timing behavior of whole systems. Classical methods of the automotive domain are often based on tests which have to be performed with real HW (e.g. evaluation boards) or cycle accurate simulators.

The limitation of this approach is that one has to cover all or at least representative system states to perform a useful prediction for the whole system. In practice this is often impossible or too time consuming. Other limitations are the preconditions due to the fact that we have to perform an architecture exploration very early in the development process. Therefore as much as possible timing predictions for different system designs have to be performed to choose the right processor. The necessary real HW or HW models are not or only in parts available. The creation of appropriate models is no practical solution because this process is very time consuming and can only be handled by HW experts. Furthermore one need for each test run all development environments (SDE; compiler, assembler, linker suite) of the chosen HW components and a suitable measurement environment. Purchasing all SDEs is cost intensive and building/running the measurement environment increases the costs additionally because of its complexity. So we

need a new approach which is cost effective and practical.

A solution can be found in the field of timing analysis which can be divided into "system analysis" and "single process analysis". System analysis is based on an abstract model of the system. It focuses on the timing and not on functional behavior. HW will not be simulated. Instead formal techniques are used which are more efficient than simulation runs to find the worst case execution time behavior of whole systems. At this abstraction level competitive access for the HW core, busses, memories, etc. can be considered. Therefore scheduling or concurrency effects caused by the OS or HW controlled operations (DMA) can be modeled. But it is important to note that this system analysis needs the single process timing of SW as input data. The single process analysis estimates the execution time of an undisturbed single process or SW component. Concurrency effects are neglected at this level. This is part of the system analysis (see above).

Suitable system level analysis already exists, such as SymTA/S (see Henia *et. al.* (2005) and Hamann *et. al.* (2006)), a tool that is commercialized by Symtavision (see Symtavision home page) and e.g. used in the automotive industry. So an appropriate method to predict the single process time in the context of early design phases is needed. The Wormhole project addresses this challenge. The approach combines simulation techniques and statistical methods and will minimize the necessary HW expert knowledge and the use of real HW or cycle accurate simulators.

The paper is organized as follows. Related work is presented in *Chapter 2*. Details of the Wormhole principles are described in *Chapter 3*. After that we explain our experiments and discuss the results (*Chapter 4*). The paper ends with a conclusion (*Chapter 5*).

2. RELATED WORK

There are some approaches available which are targeted to prediction of SW execution time with the requirement to cover most of the timing influences.

Currently simulation or measurement at real hardware is the state of the art approach. The simulator which is running on a standard PC represents the functionality and timing of the HW (ISA and micro architecture). So the executable of the whole SW, e.g., different SW components and a real time operating system (RTOS) run on the virtual HW. The accuracy of the timing depends on the HW details which are modeled. The timing accuracy of the HW models ranges from counting instructions up to cycle accurate. Cycle accurate means that the states of the HW model and of the real HW are identical on clock boundaries. For more information on terms of temporal resolution see (Bailey *et. al.*, 2005). Commercial simulators using cycle accurate models and featuring high simulation speed are currently available, e.g., Virtual Platform Designer (see CoWare home page), RealView Maxsim (see ARM home page) or METeor (see Vastsystems home page). 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, architecture 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 Lazarescu *et. al.* (2000), Giusto *et. al.* (2001) and Bammi *et. al.* (2000).

SW (C code) is transformed in two ways. In the first case the target SW (C code) is compiled with the target compiler. The resulting object code is transformed to C code using functions that represent assembly code of a virtual machine with a small virtual instruction set. Each virtual instruction is annotated with fix timing information (delay). The virtual assembly code in C is compiled and linked to derive a simulator which runs on a PC. The timing is calculated by counting the delays of the executed virtual instructions as they are executed. In the second case the SW is compiled with a source annotator or "virtual compiler" which replaces the target compiler. The result and the further procedure is the same as in the first case.

The relation between annotated timing of the virtual processor HW and the real processor HW to be modeled is approximated by a regression analysis. A set of benchmarks runs on the virtual HW (compiled with the virtual compiler) and on the real HW (transformed by the target SDE).

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 benefit of that approach is that the construction of the performance model requires only little HW knowledge (first case). The limitation of this approach is that the abstract model of the real HW is very simple. So the inaccuracy will be high in case of a HW with a complex ISA, micro architecture and memory model (for more details regarding HW principles see Hennessy & Patterson (2006)). Additionally the second case cannot consider adequately target compiler effects or rather its optimizations (for more details regarding compiler techniques see Aho *et. al.* (2006)).

A comprehensive methodology for SW performance estimation can be found in Bontempi & Kruijtzer (2004). All necessary steps are described in an abstract form. The principle of performance modelling is based on identifying a suitable set of parameters (signature) which have an impact on the timing and on finding a relation between signature and performance (performance model). Additionally a special non-linear performance model estimation technique is introduced - lazy learning. The principle works as follows. In the training phase the signature is collected and stored with the corresponding execution time in a database. During the estimation phase the signature values of the application SW are generated. Then a set of neighbours is calculated based on the database of benchmarks. This means that training cases with similar signature values are selected. Their relevance is weighted regarding some suitable criterion. After that a regression function has to be chosen for this local subset (here: linear regression based on least square error) and the regression values are computed. Finally the calculated weights are combined linearly with the application SW signature.

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 difficulty is the necessary amount of stored data. The signatures and execution times of each benchmark must be stored. This local approach consumes much more storage space than global ones which need only one performance model.

3. WORMHOLE

The objective of Wormhole is to assist the developer with predicting the performance of SW with an acceptable accuracy. It extends over previous work, in particular the approach of identifying suitable parameters ("signature") reflecting the main timing influences and finding a relation between the execution time and the signature (performance model). Although the significance of the signature is noted no systematic procedure or framework is available so far which is a major obstacle to its practical application. This is the point where Wormhole begins. Especially the customization effort which is necessary to adapt the framework to different processors is minimized.

Wormhole puts simulation and statistical analysis together. It uses traces of executed instructions which are created by fast instruction set simulators (ISS). This kind of simulator is chosen because it is easily available for most embedded architectures. ISSs are typically included in the software development environment (SDE) of the SW engineer to verify the functional correctness of the SW. A further benefit is that all timing influences of the compiler optimization are visible at this level. Micro architectural features are missing and, therefore, the timing has to be modeled separately. The model is based on a signature which consists of a HW dependent and a generic part. The generic part is constructed by a generic simulation model which represents classical timing influences like data or control hazards. Due to this abstraction one can implement HW features independently of the concrete HW that is used. So the implementation can be reused for other processors without any adjustments. During training runs the signature is related to execution time and a performance model is created. More details are presented in the following.

3.1 Training phase

Figure 1 shows the principle of the training phase. The executables of different benchmarks run on an ISS which is part of a standard SDE. Much information of the instruction trace is

extracted and translated to an intermediate format (instruction transformation). This intermediate format is the foundation for modeling common timing influences independently of the used processor. At the moment the intermediate format consists of the instruction type, read registers (numbers not contents), written registers (numbers not contents), instruction address and data address (if applicable).

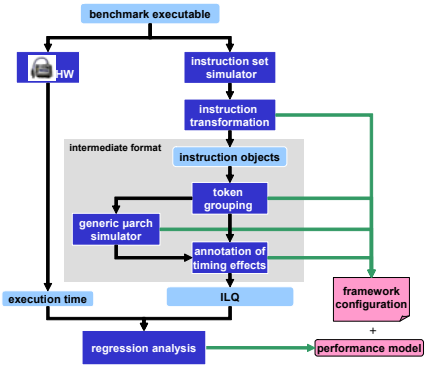

Fig. 1. Training phase

If a priori knowledge about the timing is available it is recommended to use it here (but it is not mandatory). The reason is that the number of parameters can increase extensively and boosting the required number of test cases or benchmark runs. More details are introduced later.

This generated object stream is forwarded to a generic simulator. Based on information of the intermediate format different kinds of micro architectural effects which have an impact on the timing can be detected, e.g. control and data hazards. The results are added to the affected instruction object and the unnecessary information is deleted (read/write register name and instruction/data address). The new object that encodes to an instruction and its effects is defined as instruction level quantum (ILQ). To capture the timing behavior of complex HW (e.g., pipelined processors with variable stage timing) it is often necessary to consider the history of instructions. Therefore the framework can capture the history of instruction objects. To simplify matters this kind of extension is out of scope in this paper. The ILQs are the basis for the performance model creation and are represented by *Formula 1*.

$$\vec{x} = (x_1, x_2, ..., x_n)^T; \quad (1)$$

$x_i \mathrel{\hat{=}}$ sum of occured ILQ_i

$n \mathrel{\hat{=}}$ number of different types of ILQs

In addition, the SW runs on real hardware or on a cycle accurate simulator where its execution time is measured. This step has to be performed only once per benchmark. The relation between the ILQs and the execution time is calculated by a regression analysis. Linear and non-linear models are available. Linear models are more concrete than non-linear models. Their results can be better interpreted referring to the real HW functionality. Then the relation between the measured time and the ILQs can be formulated as shown in *Formula 2*. It is called performance model.

$$t_{SW} = \vec{x}^T \cdot \vec{\alpha} = \sum_{i=1}^{n} x_i \cdot \alpha_i \quad (2)$$

$t_{SW} \mathrel{\hat{=}}$ estimated execution time of SW

$n \mathrel{\hat{=}}$ number of ILQs

$\alpha_i \mathrel{\hat{=}}$ weight of ILQ_i

The weights of the ILQs have to be calculated. For this purpose a set of m benchmarks is used to create pairs which consist of a signature vector and the corresponding execution time. These m pairs and the unknown weights span a system of equations (see *Formula 3*).

$$\vec{t} = \underline{\underline{S}} \cdot \vec{\alpha} \quad (3)$$

$$\vec{t} = (t_1, t_2, \cdots, t_m); \ \vec{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_n)$$

$$\underline{\underline{S}} = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix}$$

$t_i \mathrel{\hat{=}}$ execution time of benchmark i

$m \mathrel{\hat{=}}$ number of benchmarks

$n \mathrel{\hat{=}}$ number of ILQs

The unknown $\vec{\alpha}$ is calculated by minimizing the sum of the squared tolerances (see *Formula 4*).

$$\min\left(\sum_{k=1}^{m}\left(t_k - \sum_{i=1}^{n} \alpha_i \cdot x_{ki}\right)^2\right) \quad (4)$$

More details regarding linear regression analysis can be found in Fox (1997).

It is important to mention that the time and the values of the ILQs should be normalized with the sum of all occurred ILQs in the SW or benchmark. This process ensures a balanced

influence of low and high values of ILQs. If no relative values are used for $\vec{t}$ and $\underline{\underline{S}}$ benchmarks with great values will dominate the regression analysis (see *Formula 4*). So it seems more appropriate to replace the t_k of $\vec{t}$ and x_{ki} of $\underline{\underline{S}}$ with normalized values (see *Formula 5*).

$$t_k' = \frac{t_k}{\sum_{i=1}^{n} x_{ki}}; \qquad x_{ki}' = \frac{x_{ki}}{\sum_{i=1}^{n} x_{ki}} \tag{5}$$

One important requirement for solving such systems of equations is that $m \geq n$ is fulfilled. This means that at least one test case for each ILQ weight is needed. Because current ISAs are quite complex the number of different instruction types can be very high. This number has to be multiplied by the annotated parameters of the generic simulator to obtain the number of ILQ types. As a consequence the ILQ space increases to an impractical level. In practice the limiting factor is the number of available test cases or benchmark runs. For this reason the ILQs can be collected in groups. This grouping is optional and improves accuracy, but requires a qualified person to decide which instructions have the same timing behavior (e.g., instructions which follow the same path through the pipeline with the same timing per pipeline unit). Common information sources are technical datasheets.

Knowledge of the exact ILQ timing reduces the number of required test cases, too. This works as follows. The known portion of time is subtracted from the measured execution time of the benchmarks. See *Formula 6*.

$$\vec{t}' = \vec{t} - (x_{1z} \quad \cdots \quad x_{mz})^T \cdot \alpha_z \tag{6}$$

$\vec{t}_z' \triangleq$ new timing vector without timing influence of ILQ_z

$\alpha_z \triangleq$ known weight of ILQ_z

After that the known weight element α_z of the weight vector $\vec{\alpha}$ and the corresponding column of matrix $\underline{\underline{S}}$ can be eliminated. The column rank and therefore the number of necessary test cases are reduced.

Furthermore the knowledge of experts can help to decide which annotations have to be used at the beginning of the training. So the complexity can be reduced again. All this is optional.

In this context it is important to mention that the benchmarks have to stress all of theses parameters intensively (benchmark set quality). This means that the benchmarks take care that each parameter of $\underline{\underline{S}}$ should occur multiple times with significant values.

After the training phase all information which is necessary to predict the execution time of new SW without using real HW for the chosen HW/compiler combination is stored. This data includes the instruction transformation module, the configuration of the grouping module, the configuration of the generic simulator and the weight vector with the corresponding signature.

3.2 Prediction phase

Figure 2 shows the estimation process for new SW. First the developer prepares the framework with the same configuration as during the training phase. So the instructions will be processed in the exact same manner. For this purpose she uses the data which is stored during the training phase for the desired HW/compiler combination.

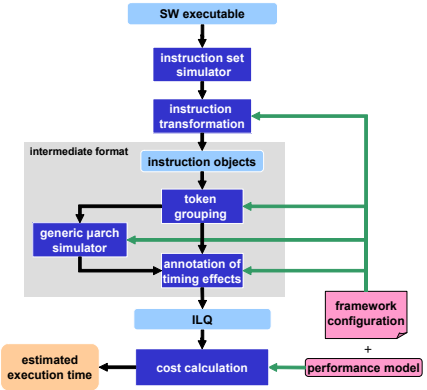

Fig. 2. Time estimation - Wormhole

If the ISS differs from the ISS used in the training phase, the instruction transformation module has to be adapted to the different output format.

Then the SW executable is executed on the ISS. The generated objects are grouped and annotated according to the results of the generic micro architecture simulator. The resulting ILQs are counted and combined with the stored

weights (see *Formula 2*). As a result one gets the estimated execution time of the SW.

4. EXPERIMENTS

To investigate and test the methodology it is evaluated in cooperation with the Ford Research Centre Aachen. We start with experiments which are performed on a Freescale HCS12 evaluation board from ELMICRO (ELMICRO, 2005). The DSE (compiler, linker, assembler) inclusively the ISS ("MIKSIM") comes from Cosmic Software (see Cosmic home page). The SW which we want to estimate is an automotive application. It is input data dependent. For each input data set an execution time prediction is performed.

4.2 Training phase

The procedure is based on the explanations in *Chapter 3*. For the training we chose a collection of benchmarks which are based mainly on C-LAB (see C-LAB home page) and EEMBC (see EEMBC home page) benchmarks (39 altogether).

We start with some adaptations of the Wormhole framework and the training phase. A new transformation module is created to handle the new ISA (especially multiword instructions) and HW status flags which are necessary to separate the different kind of instructions. The benchmarks are compiled and simulated on the board. So we obtain the measured real timing. Additionally the binary runs on the ISS. The generated trace is transformed to the intermediate format and forwarded to the grouping module. The grouping of the instruction objects and the configuration of the micro architectural simulator is based on the information of the data sheet Freescale (2003).

Due to the simple architecture of the HCS12 only the control hazard detection mechanism of the generic simulator is activated. Although the processor uses no pipeline its instruction queue has to be refilled in the case of wrong instruction fetches. This will happen if a branch is "taken" and changes the sequential execution flow. Furthermore data hazards cannot occur because of the missing pipeline. Therefore a signature consisting of eight different ILQs is sufficient.

Before we start calculating the performance model we check the quality of the benchmarks (see *Chapter 3.1*). This means that we look at the distribution of the normalized ILQ values for each ILQ over all benchmarks. This can be done easily with a scatter plot (see *Figure 3*).

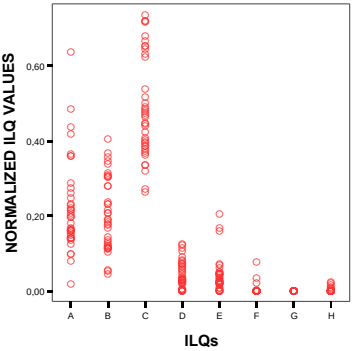

Fig. 3. Scatter plot

Each red circle represents a normalized value of a specific ILQ and benchmark. A plot (or benchmark set quality) is optimal if a lot of high values for each ILQ are available. This is necessary to give the regression algorithm a chance to calculate accurate weights (see *Formula 4*). As one can see, the distribution is not optimal. But, the benchmarks set should be sufficient (except for ILQ *G*) because we use normalized values. The normalization allows ILQs with small number of occurrences to be properly represented (see *Formula 4*). But that is not sufficient as can be seen. The ILQ *G* occurs seldom and the values are all nearly 0! So some inaccuracy is expected.

We start the estimation of the weights. The counted and normalized ILQs and the real timing are put in a regression analysis with the least square metrics (see *Chapter 3.1*) and the performance model or weights of the ILQs are calculated. We use the tool SPSS which features comprehensive statistical analysis techniques (see SPSS home page).

4.3 Results

The ILQ weights and the corresponding estimation errors of the predicted weights are presented in *Figure 4*. As one can see all ILQ weight predictions with the exception of G are very precise. The absolute estimation errors of the ILQ weights are less 0.6% except for the

seldom ILQ *G* which is predicted with an error of over 225%!

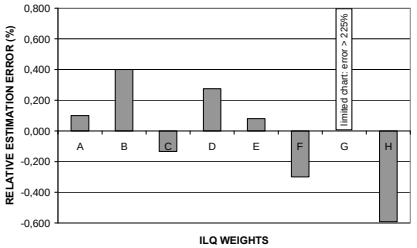

Fig. 4. Estimation error of ILQ weights

But, the absolute estimation errors of the application SW execution times (different input data sets) are **less 0.4%** (see *Figure 5*). By the way, the errors are exclusively negative because of measurement errors that occurred while the real execution times of the SW runs are captured. Due to the setup the measured execution times are slightly greater than the actual execution times.

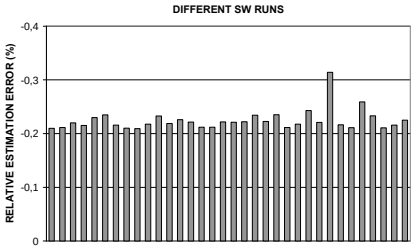

Fig. 5. Estimation error of the SW runs

The reason for the high accuracy in spite of the ILQ *G* problem is that the ILQ *G* does not occur in the application SW runs. Therefore no negative influence can occur. But, if other SW is estimated it will be important that the proportion of ILQ *G* will be checked. A high estimation accuracy can only be achieved if the proportion of ILQ *G* is negligible.

It is important to note that the estimation errors of the ILQ weights can only be calculated in an experimental setup where the reference timing is known a priori (clarifying the methodology). In practice these weights are unknown and shall be estimated. So the performance model has to be verified by calculating the execution time estimation errors of benchmarks during the training phase, e.g., using the "leave one out" technique.

The evaluation has shown that the approach works fine for simple processors. Wormhole is flexible and can be adapted easily to other architectures. At the moment we are investigating an ARM9 core (no thumb, no floating point, simple memory model) which is more complex than the HCS12. Pipeline interlocks due to data and control hazards has to be considered. The first results are promising. The SW execution time estimation error is less than 5%.

But, very complex architectures can only be considered with limited accuracy. Long histories or many necessary architectural annotations, e.g., caused by caches, complex branch strategies, massive superscalar behavior, etc. lead to a parameter space explosion. This amount of parameters is necessary to form a suitable analysis input model and finally a performance model. Therefore these effects can be recognized in practice only in parts or rather for HW with short instruction history dependencies. However, most processors used in, e.g., automotive applications do not expose such complex behavior.

5. CONCLUSION

The Wormhole project goal was estimation support for early design space exploration, both for automotive network planning and ECU design. With an increasing industrial acceptance of formal methods in automotive system performance verification, formal techniques become also available for earlier design phases. Other than in performance verification, the architecture is not yet fully defined leaving processor architecture alternatives. As a consequence, design space exploration typically requires intensive experiments with different architectures or estimations. This is a costly and time consuming step.

At the very beginning of the design process, the software is at best incompletely available such that estimations in the form of educated guesses and abstract load level models are all that is available as input data. Deriving such data is a research topic in its own right that is not addressed here. This project assumes that software in the form of C code and test patterns are available, from the increasing body of reused software, from third party providers or from

early design of performance critical software parts. For these SW components, performance estimations shall be derived with high accuracy without the need to run through a prototyping or simulation task. Instead, a qualification process is assumed that is executed once per processor or once per processor and load characteristic. That qualification process is expected to be outsourced to a service provider.

Results for simpler architectures and partly available SDE show high estimation accuracy in the order of 1% estimation error. Due to parameter space and benchmark count limitations, complex architectures can usually not be covered. However, in many practical systems, such as in most automotive applications, high-performance processor architectures are avoided to obtain predictability and reduce cost such that the presented approach can be applied.

ACKNOWLEDGEMENT

This work is sponsored in part by the Ford University Research Program.

REFERENCES

Aho, A. V., M. S. Lam, R. Sethi and J. D. Ullman (2006). Compilers: Principles, Techniques, and Tools. Addison Wesley, 2nd edition.

ARM home page. The Architecture for the Digital…, http://www.arm.com, 2008.07.02.

AUTOSAR home page. AUTOSAR, http://www.autosar.org, 2008.07.02.

Bailey, B., G. Martin, T. Anderson (2005). Taxonomie for the Development and Verification of Digital Systems. Springer.

Bammi, J. R., E. Harcourt, W. Kruitzer, L. Lavagno, M. T. Lazarescu (2000). Software performance estimation strategies in a system-level design tool. *Proc. of the Eighth International Workshop on Hardware/ Software Codesign (CODES)*, 82-86.

Fox, J. (1997). Applied Regression Analysis, Linear Models, and Related Methods. Sage Publications.

Bontempi, G. and W. Kruijtzer (2004). The use of intelligent data analysis techniques for system-level design: a software estimation example. *Proc. Soft Computing - A Fusion of Foundations, Methodologies and Applications,* 8(7), 477- 490.

C-Lab home page. C-Lab Homepage, http://www.c-lab.de, 2008.07.02.

Cosmic home page. Cosmic Software Cross…, http://www.cosmic-software.com, 2008.07.02.

CoWare home page. The ESL Design Leader, http://www.coware.com, 2008.07.02.

EEMBC home page, EEMBC–The Embedded…, http://www.eembc.org, 2008.07.02.

ELMICRO (2005). HCS12 T-Board, Hardware-Version 1.00. User Manual.

Freescale (2003). HCS12 Microcontrollers, S12CPUV2. Reference Manual.

Giusto, P., G. Martin, E. Harcourt (2001). Reliable Estimation of Execution Time of Embedded Software. *Proc. of the conference on Design, automation and test in Europe (DATE)*, 580-589.

Hamann, A., R. Racu und R. Ernst. Formal Methods for Automotive Platform Analysis and Optimization (2006). *In Proc. Future Trends in Automotive Electronics and Tool Integration Workshop (DATE Conference)*.

Henia, R., A. Hamann, M. Jersak, R. Racu, K. Richter und R. Ernst (2005). System Level Performance Analysis - the SymTA/S Approach. *IEEE Proc. Computers and Digital Techniques,* 152(2), 148-166.

Hennessy, J. L., D. A. Patterson (2002). Computer Architecture – A Quantitative Approach. Morgan Kaufmann, 3rd edition.

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

SPSS home page. SPSS, Data Mining,…, http://www.spss.com, 2008.07.02.

Symtavision home page. Symtavision – Scheduling Analysis for ECUs, Buses…, http://www.symtavision.com, 2008.07.02.

Vastsystems home page. VaST tools and models for embedded system design, http://www.vastsystems.com, 2008.07.02.