Performance Analysis with Confidence Intervals for Embedded Software Processes

Per Bjuréus
Saab Avionics, Sweden
perb@imit.kth.se

Axel Jantsch
Royal Institute of Technology, Sweden
axel@imit.kth.se

Abstract

The choice of algorithms has a large impact on the performance of embedded real-time systems. Therefore, performance estimation of embedded software is vital in an early design phase. Consequently, high-level estimation techniques have been devised, but the accuracy of the estimations vary a lot depending on the algorithm and its context. We address this problem by proposing an estimation technique that both estimates the performance and computes the expected accuracy. The accuracy is used to provide a confidence interval to the estimated performance. The estimation framework presented in this paper has been crafted to fit with the MASCOT environment, but the underlying techniques can also be applied to other high-level design exploration frameworks.

1. Introduction

Performance estimation of high-level specifications is mostly performed in a back-of-the-envelope fashion based on using experience. This procedure either leads to unpredictable and poor accuracy or involves many experienced engineers and time consuming discussions and studies. Hence, estimation tools have been developed, e.g. Cadence VCC [6], the SPI workbench [1], and the Polis codesign tool from UC Berkeley [2]. Using tools is much faster and a large number of different solutions can be explored in short time. Unfortunately, the accuracy of the estimations vary widely and, even more important, it is largely unknown. This is the problem that we address.

Under the assumption that the system is specified as a collection of concurrent processes that are invoked sporadically, the estimator computes the execution time of invocations and provides a confidence interval. The technique uses an execution trace, generated by simulation, which is hierarchically decomposed. The decomposition exposes the source of uncertainty, and enables the user to focus on profiling and characterization of components and operations that contribute most to the inaccuracy. Hence, the estimation can be gradually improved until a satisfactory level of accuracy has been reached.

Performance analysis and estimation can be divided into static and dynamic techniques. Static techniques [1][4][8] are concerned with the analysis of a specification without simulating or executing the specification. It is often employed for worst-case analysis and is suitable for finding corner cases that are hard to cover with simulation or execution. Dynamic techniques [2][6][7] on the other hand, are concerned with the analysis of run-time behavior, and relies on test vectors and input scenarios. Dynamic techniques are preferred when the system performance is heavily dependent on the input data, and in cases where the average performance is more relevant than worst-case behavior.

Besides classification into static and dynamic approaches, the level of abstraction also characterizes different performance estimation techniques. Low-level estimation [5] considers compiled and optimized machine code and employs detailed models of the processor architecture and memory hierarchy. Low-level estimation can thus accurately account for operating system, pipelining and cache effects. High-level estimation [9][10] on the other hand deals with high-level specifications and employs a very abstract model of the architecture. Both low-level and high-level estimation fulfills important needs in system design; high-level techniques are excellent for early feasibility studies, and low-level techniques yield unprecedented accuracy in a late design phase.

Our method is a high-level, dynamic estimation technique. However, it not only estimates the performance but also the accuracy of the result. In this respect it is complementary to most other high-level estimation techniques. The main features are the following.:

- Performance estimation is performed on an arbitrary *high-level specification*, where the actual code or RTL description is largely unknown. This means that we have little hope of getting high accuracy, instead, we try to compute the accuracy, and provide information about the origin of potential inaccuracy.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

- Our approach is *trace-based*, which means that the simulation needs only be run once in "trace-mode", where all operations are recorded. This decouples the behavioral specification from the estimation and speeds up the estimation significantly when compared to annotated performance simulation.
- We employ a *hierarchical decomposition* of functions, which gives the developer a quick and dirty first approximation but allows detailed profiling and characterization of complex algorithms. This also enables designers to collect and post knowledge and experience into the framework for future use and reuse.
- The estimation is based on *statistical models*, which is the key to accuracy analysis.

2. Framework and Workflow

The estimation framework and workflow complements the MASCOT [12] modeling environment. The workflow is based on a separation of the behavioral specification from the architectural model (Figure 1).

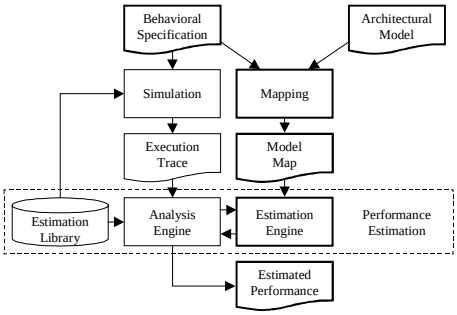

Figure 1. Estimation workflow

The behavioral specification, simulation, and execution trace is discussed below. The architectural model and the mapping is described in section 3.

Although important, we do not address communication in this paper. However, we include communication primitives (e.g. signals, channels, transactions, etc.) in the definitions that follow for completeness.

2.1. Behavioral Specification and Simulation

The behavioral model consists of a set of concurrent communicating processes. A process consumes input signals, performs necessary computations and potentially produces output signals. Communication is carried out asynchronously with unbounded FIFO buffers.

Definition 1. A *behavioral specification* is a network of processes represented as a tuple $B=(P, C, S, a)$, where P is a set of processes, $C \in P \times P$ is a set of channels, S is a set of signals, and $a: S \rightarrow C$ is a function mapping signals to channels.

Execution and communication is supposed to be ideal in the sense that it takes zero time. As a consequence, time is only advanced by the simulation stimuli or internal timers. This implies that the external stimuli or an internal timer may trigger an arbitrarily long sequence of internal events, which must be completely resolved before the next external event or timer can be processed. During simulation all process execution and signal communication is recorded in an *execution trace*. It is left to the designer's discretion to ensure that the simulation is realistic.

2.2. Execution Trace

Using an execution trace decouples the behavioral specification from the estimation environment. There are two reasons to do this. First, the specification that we primarily target is heterogeneous, e.g. a mixed Matlab-SDL model, but a homogeneous input to the estimator is preferred since it is easier to process. Second, we envision an estimation framework that can be used with other system specifications, which requires a well defined intermediate format that links the specification and the estimator. Note that this cannot be achieved easily with annotated performance simulation.

There are two atomic elements of the execution trace, the operation and the transaction, as defined below.

Definition 2. An *operation* is a five-tuple: $op=(i, id, pl, ol, r)$, where $i \in \mathbf{N}$ is the instance count, $id \in ID$ is a unique name, $pl=\{p_1,p_2,\ldots,p_n\}$, $n \in \mathbf{N}$ is a parameter list, $ol=\{op_1,op_2,\ldots,op_n\}$, $n \in \mathbf{N}$ is an operation list, and $r \in \mathbf{N}$ is the repeat count.

The operation list component *ol* allows us to build hierarchical operations of arbitrary depth. The instance count *i* determines the number of instances of the operation that can be performed in parallel, and the repeat count *r* establishes the number of times the operation must be performed in sequence. This allows a crude estimation of parallelism for devices where operations may be performed concurrently, e.g. in SIMD architectures. For sequential processing, such as software executed on an instruction set processor (ISP), the two numbers *i* and *r*, are simply multiplied to give the total count of the operation.

Definition 3. A *transaction* is a pair: $tr=(n, w)$, where $n \in \mathbf{N}$ is the number of data tokens transmitted, and $w \in \mathbf{N}$ is the bit width of each token.

Definition 4. An *execution trace* is a list of invocations: $et=\{I_1, I_2, \ldots, I_n\}$, $n \in \mathbf{N}$. An invocation is a tuple $I_k=(t_k, id, sl, el)$, where $t_k \in \mathbf{R}$ is a time-stamp, $id \in P \cup S$ uniquely identifies a process or signal, $sl=\{s_1, s_2, \ldots, s_n\}$, $n \in \mathbf{N}$, $s_k \in S$ is a list of triggering signals, and

$$el = \begin{cases} \{op_s, op_2, \ldots, op_n\} & \text{if} \quad id \in P \\ \{tr_s, tr_2, \ldots, tr_n\} & \text{if} \quad id \in S \end{cases}$$

is either a list of operations or a list of transactions.

The time stamps of the invocations are non-decreasing, i.e. $t_k \leq t_{k+1}$. Since an external event or internal timer can trigger a sequence of invocations with the same time stamp, the list of triggering signals is used to resolve data dependencies.

In MASCOT we generate execution traces both from SDL and Matlab. The SDL simulator allows us to generate an execution trace simply by switching on the logging function. In Matlab, we have devised a technique to generate an execution trace that relies on object-orientation and overloading of operations, which then write the trace information to a file. After simulation the generated files are merged and filtered to create the execution trace.

3. Architectural Model

The architectural model consists of a set of connected processing and communication units.

Definition 5. An architectural model is a triple *am=(X, T, E)*, where *X* is a set of processing (execution) units, *T* is a set of communication (transmission) units, and $E \subseteq X \times X \cup T \times T \cup X \times T \cup T \times X$ is a set of directed edges connecting the nodes.

A processing unit is modeled as an abstract machine characterized by a *virtual instruction* (VI) set. The abstract machine can not perform any computation, but only models the performance of the unit.

Definition 6. A *processing unit* is a triple $x=(V, M, \Delta)$, where *V* is a virtual instruction set, *M* is a macro instruction set, and Δ is a set of instruction profiles. An instruction profile $\delta \in \Delta$ is a function $\delta: V \cup M \rightarrow \mathbf{R} \times \mathbf{R}$ such that $\delta(op)=(\mu, \sigma^2)$, where $op \in V \cup M$, and $\mu \in \mathbf{R}$ and $\sigma^2 \in \mathbf{R}$ associates virtual instructions with an expected execution time and variance.

Selection of a VI set is a delicate task, where the goal is to select instructions that are abstract enough to support a large variety of processing units, but specific enough to differentiate between distinct types of instructions. A complete treatment of the selection of a VI set is not within the scope of this paper. The set of execution time distributions is called a VI *profile*. The selection of a proper VI profile can be done manually or automatically e.g. with signatures and discriminating functions [3].

3.1. Processor Characterization

Processor characterization can be performed analytically or empirically. The analytical approach is based on datasheet analysis [3] and requires no compilation or execution. Although a range of execution times can be derived with datasheet analysis, it is hard to capture the statistical distribution of VI execution time that reflects compiler optimizations and application domain. The empirical approach on the other hand is based on benchmarks [11] that are compiled and analyzed. This approach readily captures the statistical distribution but requires representative benchmark suites and quite a bit of manual interaction.

The VI set models the operations of generic processors, but we also must be able to capture processor specific features. A typical example is the MAC operation of a DSP processor which comprises a MUL and an ADD instruction. A similar situation may arise when the user has access to the execution time of a specific function, e.g. as provided by the processor manufacturer. In both cases, the user wants to add instructions to the processor model without ruining previous modeling efforts and without modifications that prevents future design exploration. This is supported by the *extended instruction* (EI) set. To use extended instructions, the behavioral specification is annotated with trace calls. Note that adding an extended instruction to one processor model does not require changes in other processor models. Note also that introducing trace calls in the behavioral model does not change the behavior.

3.2. Mapping

The behavioral model is mapped to the architectural model through a pair of mapping functions:

Definition 7. A *model map* is a pair *mm = (pm, sm)*, where $pm: P \rightarrow X$ maps processes to process units, and $sm: S \rightarrow T$ maps signals to communication units.

4. Performance Estimation

The estimation workflow in Figure 1 shows that the performance estimator in the dashed box has been divided into three parts: The estimation engine, the analysis engine, and the estimation library. The division between estimation engine and analysis engine allows us to choose different estimation engines for different purposes while maintaining the analysis engine intact. The estimation engine is closely coupled with the architectural model and dictates what we estimate. The estimation library is used to collect static operations, which unburdens the execution trace and simplifies macro instruction identification. The framework and performance estimator has been implemented as a tool called *EstiMate*.

4.1. Estimation Procedure

The software performance estimator consists of the analysis engine and the estimation engine. It uses the execution trace and a model map to compute the estimated performance. The estimator behaves as follows:

1. The analysis engine reads the execution trace, and for each invocation in the trace performs steps 2 through 9 below.
2. The analysis engine uses the model map to figure out which processor the invoked process runs on, and the proper estimation engine is called.
3. The estimation engine traverses the invocation operation list, and for each operation performs steps 4 through 8 below.
4. If the operation encountered is found in the VI set or EI set of the processor model, the mean and variance for the operation is recorded.
5. If the operation is not found, but contains an operation list, the operation list is expanded and step 4 is repeated for each operation in the list.
6. If the operation is not found, and cannot be expanded, the estimation engine queries the analysis engine for an estimation library equivalent.
7. If the analysis engine receives a library query, it performs a lookup, and expands the operation with the equivalent operation list. Back to 4.
8. When the entire invocation has been traversed, the invocation, annotated with execution time mean and variance, is returned.
9. The analysis engine computes the overall execution time for the invocation, and continues from step 1 until all invocations have been read.

4.2. Probabilistic Analysis

The probabilistic analysis takes place when the analysis engine has traversed the execution trace according to the algorithm in section 4.1.

To estimate the execution time of an invocation, we may treat each operation in the execution trace as a stochastic variable, denoted ξ_k^{op}, where $op \in VI \cup EI$ identifies the instruction, and $1 \le k \le n_{op}$, where $n_{op} \in \mathbf{N}$ is the total number of times that the operation *op* is executed in the invocation. The total execution time of the invocation will thus be a stochastic variable *t*:

$$t = \sum_{\forall op} \sum_{k=1}^{n_{op}} \xi_k^{op} \tag{1}$$

Consider the inner summation, where the execution times of individual operations are added:

$$t_{op} = \sum_{k=1}^{n_{op}} \xi_k^{op} = \xi_1^{op} + \xi_2^{op} + \ldots + \xi_{n_{op}}^{op} \tag{2}$$

Theorem 1. *Central limit theorem (variant).* Let $E[\xi_k^{op}] = \mu_{op}$ and $Var[\xi_k^{op}] = \sigma_{op}^2$. If n_{op} is large and the stochastic variables ξ_1^{op}, ξ_2^{op}, ... are sufficiently independent, the sum of the stochastic variables, t_{op} is approximately normal distributed:

$$t_{op} \text{ is } N(n_{op}\mu_{op}, \sigma_{op}\sqrt{n_{op}}) \tag{3}$$

Now, consider the outer summation of (1) where the sums of execution times for different operations are added:

$$t = \sum_{\forall op} t_{op} \tag{4}$$

Remembering that each t_{op} is normal distributed, and assuming that the stochastic variables t_{op} are independent, we apply the addition theorem for normal distribution:

Theorem 2. *Normal distribution addition theorem.* If ξ_1 is $N(\mu_1, \sigma_1)$ and ξ_2 is $N(\mu_2, \sigma_2)$, then:

$$\xi_1 + \xi_2 \text{ is } N(\mu_1 + \mu_2, \sqrt{\sigma_1^2 + \sigma_2^2}) \tag{5}$$

This yields the distribution for the total execution time:

$$t \text{ is } N(\sum_{\forall op} n_{op}\mu_{op}, \sqrt{\sum_{\forall op} n_{op}\sigma_{op}^2}) \tag{6}$$

Note that we have made two rather strong assumptions, first we have assumed that n_{op} is large, and second that operation execution times are independent. We address these issues in the following paragraphs.

When n_{op} is not large, the stochastic variable t_{op} will not be normal distributed. However, as long as the execution times of different operations are independent (which is a reasonable assumption), we can use the addition theorem for stochastic variables:

Theorem 3. *Addition Theorem.* Let ξ and η be two independent stochastic variables, then $E[\xi+\eta] = E[\xi]+E[\eta]$, and $Var[\xi+\eta] = Var[\xi]+Var[\eta]$.

Since the stochastic variables ξ and η are not normal distributed, the sum will not be normal distributed either. When determining a confidence interval for the sum, we have to resort to a more conservative method known as Chebyshev's inequality.

Theorem 4. *Chebyshev's inequality.* If ξ is a stochastic variable with $\mu = E[\xi]$ and $\sigma^2 = Var[\xi]$, then

$$P(|\xi - \mu| \ge k\sigma) \le \frac{1}{k^2}, \quad k > 0 \tag{7}$$

If the execution times of different operations in the execution trace are not independent, we are not allowed to

add the variance according to the addition theorem. Consider an execution trace where the same operation is executed several times, either due to the instance and repeat counts, or as a result of being located inside a loop. That operation will not likely change its execution time between different executions. The variance must be computed according to the following theorem:

Theorem 5. *Constant factor*. If ξ is a stochastic variable, then $Var[a\xi] = a^2 Var[\xi]$.

5. Experiments

We have carried out two experiments to investigate weather the proposed method has any relevance in reality.

The experimental setup is shown in Figure 2. The behavioral specification that we used is the *test1* program from [10], and an image rotation program *imrotate*, both of which were translated both to Matlab and C code.

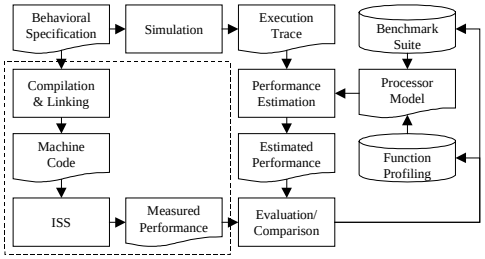

Figure 2. SW Estimation experimental setup

The Matlab code was simulated, generating an execution trace, which was fed to *EstiMate*. The C code was compiled for an ARM7TDMI processor, and executed with an instruction set simulator (ISS). The estimation engine returned an execution time interval with a confidence degree, which was compared with the measured performance from the ISS.

The ARM7TDMI processor was modeled with a virtual instruction set shown in Table 1.

Table 1. ARM7TDMI Virtual Instruction Set

Instruction	μ	σ^2
ADD	1.500000	0.500000
LOOP	5.000000	0.500000
LE	5.500000	0.500000
FLOOR	1.000000	0.100000
MUL	1.500000	1.000000
SUB	1.500000	0.500000
GT	5.500000	1.000000
DIV	1.500000	1.000000
subsref	2.000000	0.200000

The model was derived from a small part of the same code that was used to estimate the execution time, which explains the extremely small variance.

We ran the experiment 16 times, varying the number of loop iterations of the four loops of the program (L1-L4). The confidence degree was set to 95%, and the result using a two-sided confidence interval based on a normal distribution is shown in Figure 3. The vertical bars represent the estimated interval, and the diamond shows the measured execution time.

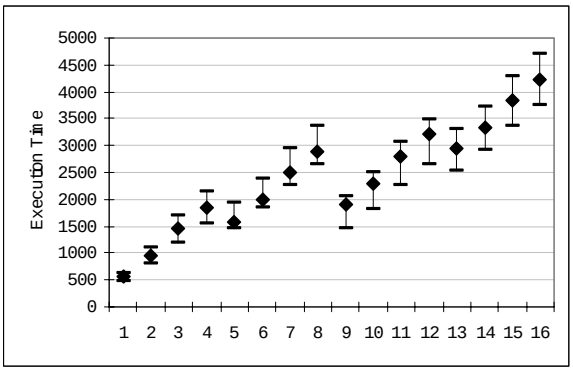

Figure 3. Outcome of test1 experiment

For the second example (image rotation) we had to derive a different processor model taking floating point emulation into account. The Virtual Instruction set was enlarged to contain 16 different virtual instructions. An additional 3 instructions were modeled using the Extended Instruction set and 11 instruction were added to the estimation library to model special Matlab functions. In the image rotation program, the number of independently executing operations was relatively small, and therefore a confidence interval based on Chebyshev's inequality was selected. The measured execution time was well within the estimated interval. Figure 4 shows a screenshot from the execution time analysis.

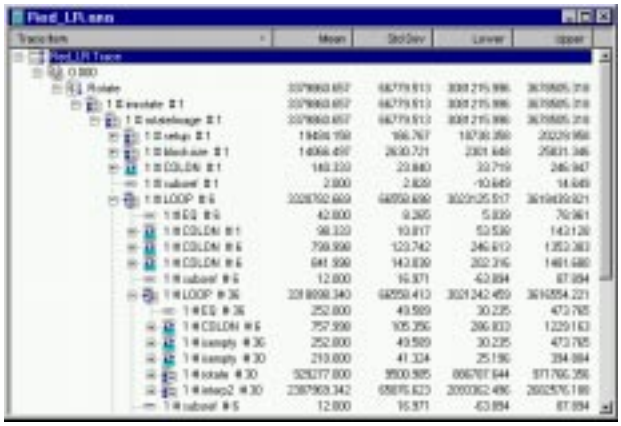

Figure 4. Execution time analysis of imrotate

Note that the relative accuracy of the estimation increases when the number of independent variables decreases, see Table 2. At the same time, we observe a larger relative error in the estimated execution time.

Table 2. Relative error and estimated accuracy

Segment	Measure	Estimate	Error	Accuracy
imrotate	3220928	3379861	4,9%	±8,4%
blocksize	20552	14066	-31,6%	±83,6%
initbounds	1295	1215	-6,1%	±29,6%
init_rows	152	133	-12,3%	±69,2%
init_image	17402	7882	-54,7%	±149,1%

The accuracy in the table above is the estimated interval relative to the estimated execution time.

6. Conclusion

We have presented a novel technique for dealing with accuracy in high-level performance estimation. The technique resides in a design exploration framework, which provides a stable environment for tools and methods. Preliminary results have been reported, demonstrating the workflow, and showing that the technique has good potentials, encouraging further development.

It is important to remember what one is actually estimating, especially when dealing with a high level specification like we do. There are innumerous possible implementations of the algorithms we estimate, some clever and some not. The estimated execution time that our estimator delivers corresponds to one possible implementation of the specification. Therefore, the trace macro library plays an important role giving the designer an opportunity to model an algorithm in an abstract way. Even though the accuracy may seem to be of limited usefulness, the estimator clearly shows where the bottlenecks may show up, and gives a good idea about what constraints are reasonable for further development.

We are continuously working larger example applications, and we expect to apply the estimation technique on the full-blown Digital Receiver [13] in near future.

7. References

[1] Marek Jersak, Dirk Ziegenbein, Fabian Wolf, Kai Richter, Rolf Ernst, “Embedded System Design using the SPI Workbench”, *Proceedings of the 3rd International Forum on Design Languages*, Tübingen, Germany, September 2000.

[2] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Key Suzuki, Bassam Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, 1997.

[3] Paolo Giusto, Grant Martin, Ed Harcourt, "Reliable Estimation of Execution Time of Embedded Software", *Proceedings of the Design, Automation, and Test in Europe (DATE) Conference 2001*, Munich, March 2001.

[4] Luca P. Carloni, Alberto L. Sangiovanni-Vincentelli, "Performance Analysis and Optimization of Latency Insensitive Systems", *Proceedings of the 37th annual Design Automation Conference (DAC)*, Los Angeles, 5-9 June 2000, pp. 361-367.

[5] Mihai T. Lazarescu, Jwahar R. Bammi, Edwin Harcourt, Luciano Lavagno, Marcello Lajolo, "Compilation-based Software Performance Estimation for System Level Design", *Proceedings of the IEEE International High-Level Design Validation and Test Workshop*, Berkeley, CA, USA, 8-10 November 2000, pp. 167-172.

[6] Jwahar R. Bammi, Edwin Harcourt, Wido Kruijtzer, Luciano Lavagno, Mihai T. Lazarescu, "Software Performance Estimation Strategies in a System-Level Design Tool", *Proceedings of the Eighth International Workshop on Hardware/Software Codesign. CODES 2000*, San Diego, CA, USA, 3-5 May 2000, pp. 82-86.

[7] Jie Liu, Marcello Lajolo, Alberto Sangiovanni-Vincentelli, "Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator", *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES /CASHE'98)*, Seattle, WA, 15-18 March 1998, pp. 65-69.

[8] Maher Rahmouni, Ahmed A. Jerraya, "Formulation and Evaluation Of Scheduling Techniques For Control Flow Graphs", *Proceedings of EURO-DAC European Design Automation Conference*, Brighton, UK, 18-22 September 1995, pp. 386-391

[9] Subhrajit Bhattacharya, Sujit Dey, Franc Brglez, "Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications", *Proceedings of 31st ACM/IEE Design Automation Conference*, San Diego, CA, USA, 6-10 June 1994, pp. 491-496.

[10] Kamal S. Khouri, Niraj K. Jha, "Clock Selection for Performance Estimation of Control-Flow Intensive Behaviors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 1*, January 2001, pp. 158-165.

[11] Umesh Krishnaswamy, Isaac D. Scherson, "A Framework for Computer Performance Evaluation Using Benchmark Sets", *IEEE Transactions on Computers, Vol. 49, No. 12*, December 2000, pp. 1325-1338.

[12] Per Bjuréus, Axel Jantsch, "MASCOT: A Specification and Cosimulation Method Integrating Data and Control Flow", *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*, Paris, France, March 2000.

[13] Per Bjuréus, Fredrik Hoffman, "Modeling a Digital Receiver with MASCOT", *Proceedings of the Designer's Forum at Design, Automation & Test in Europe Conference (DATE)*, Munich, Germany, March 2001.