Architecture-Level Performance Evaluation of Component-Based Embedded Systems

Jeffry T Russell, Margarida F Jacome
Electrical and Computer Engineering, University of Texas at Austin
jeffry@mail.utexas.edu, jacome@ece.utexas.edu

ABSTRACT

A static performance evaluation technique is proposed to support early, architecture-level design space exploration for component-based embedded systems. The novel contribution is the use of a designer-specified evaluation scenario to identify a characteristic subset of system functionality that serves as a context for a rapid performance evaluation between candidate architectures. Fidelity is demonstrated with a case study that compares performance estimates of several candidate architectures to measurements from respective implementations.

Categories and Subject Descriptors

J.6 [Computer-aided engineering]: Computer-aided design; B.8.2 [Perf. Analysis and Design Aids]: Software analysis.

General Terms

Performance, Design.

Keywords

Performance evaluation, architecture-level, design space exploration, component-based, embedded system, scenario.

1. Introduction

A real-time embedded system interacts with its environment constrained by a set of timing deadlines. Some constraints are considered "hard", while others are "soft", meaning the timing deadlines may be violated in the context of a trade-off like cost, e.g. an MPEG decoder may miss an occasional video frame. A performance evaluation for these systems focuses on whether timing constraints are met, which requires an analysis of all system functionality to find the *worst case* functional paths.

In contrast to real-time systems, this paper considers a class of embedded systems for which performance is a figure of merit. Performance is still important, but it is a perceived quality of the system, rather than a necessary condition for a correct design. Thus, as compared to real-time embedded systems, the design approach differs in that performance is a distinguishing figure of merit among feasible designs and comparison between such designs is based on a *typical case* subset of functionality.

Component-based designs are beneficial in terms of shorter development cycles, reduced cost, and better maintainability [4][14]. Component-based embedded systems favor architectures consisting of predefined subsystems, packaged as components, rather than custom-specified subsystems. Hardware components such as CPUs or memories provide structural resources, software components provide pure functionality, and some components, such as I/O controllers, provide functionality bundled with resources.

For example, consider the embedded system design of a TCP interface using components such as high-performance CPUs and memories. Though technically a soft real-time system, with a 3 second default packet timeout [3], a practical approach may assume that the components allow the protocol constraint to be easily satisfied and therefore treats performance as figure of merit during design.

Furthermore, consider the use of a software component that is a TCP protocol library. Such a component supports a reliable communication link and therefore includes exception handling and error recovery code that is not activated during typical case operation. At a designer's discretion, an evaluation scenario would consider the functional paths activated by the error-free exchange of a 64KB buffer, thereby removing from consideration many feasible, yet atypical paths, related to error recovery.

Since components are often complex and include a significant amount of functionality, a designer faces the challenge of identifying the relevant, typical case subsets of functionality, called *evaluation scenarios*. A performance evaluation technique based on such an evaluation scenario is useful for a rapid, comparative assessment of candidate architectures during design space exploration.

This paper proposes a performance evaluation technique to support architecture-level design space exploration of component-based embedded systems that emphasizes relative fidelity over accuracy of individual estimates. Specifically, two key problems are addressed: (1) how to define and extract an evaluation scenario, and (2) how to evaluate the system performance based on a scenario.

The novel contribution of the proposed performance evaluation is the use of a designer-specified scenario to set the context for a rapid performance evaluation of distinct, component-based candidate architectures.

DAC 2003, June 2-6, 2003, Anaheim, California, USA.

The approach focuses on an interactive definition of evaluation scenarios through incremental refinement of a functional specification to identify control flow paths corresponding to typical case behaviors. This challenging task is assisted by a semi-automated technique that can be applied simultaneously to software and hardware components. From each evaluation scenario, a pseudo-trace is statically generated, and pertinent architecture features are distilled into operation execution costs that are used to estimate an execution time.

2. Related Work

Design space exploration research can be classified as addressing aspects of representation, estimation, and exploration algorithms [11]. Our proposed technique supports estimation by providing a rapid performance evaluation of component-based architectures suitable for use in an existing framework that generates candidate architectures.

Many performance estimation techniques used in real-time systems are based on a course-grained model in which the atomic element is a software *process*. This work considers a lower level of detail, e.g. instructions or operations.

Within this scope of related work, the proposed performance evaluation technique is novel in the sense that it is an abstract, static model that reports a value for *typical case* operation (sometimes called an *average* case). Generally, static models have been used to determine bounds on performance, and most estimates that return typical values are based on dynamic models.

Static, abstract performance estimation models are present in codesign methodologies to facilitate quick computation, e.g. POLIS [15], COSYMA [1], and TOSCA [1]. These environments rely upon a *restricted* specification language, which is biased towards a custom-specified (or generative) design approach, rather than the component-based approach used in our technique.

Several performance estimation techniques use a more detailed, static analysis model to conservatively estimate a performance bound, e.g. the tool Cinderella [9] implements a static analysis technique that implicitly considers all possible execution paths to find the extreme basic block execution counts. The execution time of each basic block is derived from databook values.

Dynamic estimation techniques are essentially based on a simulation of a system, from which performance is measured, e.g. Ptolemy [10][16]. Such approaches are poorly suited for design space exploration because of the need to construct a virtual system, the hand coding of abstract models, or the time required for simulation.

A few recent publications have similarly reported estimates for a typical case based on a static model. Gupta et al. proposed a static model generated by profiling a program and relying on databook values to determine operation costs [6]. Lahiri et al. focus on bus communication using a previously captured execution trace [8], but do not consider other architecture features such as variations in the configuration of busses and memories. The abstract s-graph representation in POLIS can be used to estimate typical values, where profiling is used to determine the typical case path through the graphs [15].

Though these approaches operate on static models, they all require an initial dynamic analysis to characterize the functional paths. In contrast, our proposed technique assists a designer in the specification of the evaluation scenario using only static models.

3. Background

Component-based design is a common topic in recent literature, but the term "component" usually implies a software member and treats hardware as a fixed execution platform [2][5][14]. This paper uses *component* to describe hardware or software members whose specification is fixed, though possibly parameterized. Our precise definition of component is deferred until the features of architecture representation are explained.

3.1 Representation Model

To support the proposed evaluation model, a minimal representation model is described. An *architecture* is an abstract representation of a design that is expressed as a topology or configuration of subsystems. An architecture is decomposed into two distinct, but related, topologies: functional and structural.

In a *functional architecture*, subsystems specify functionality and connections indicate flow of control linkage. For convenience, a subsystem in the functional architecture is called a *module*. The functional architecture specifies a set of (possibly) concurrent, sequential processes, where each process is specified by one or more modules. We assume each module is organized as one or more *procedures*, and each procedure is specified using an imperative language, i.e. as a sequence of operations. The *operations* are the primitives of functionality, and the level of abstraction can vary form source code operations to assembly instructions or their hardware counterparts. The example functional architecture in Figure 1(a) demonstrates a single software process specified with several modules. The connections between the modules, the *linkage*, express call/return control flow relationships. A module describing a separate hardware process, labeled "I/O Functionality", is also shown. The important aspect of the functional architecture is that flow of control can be followed between modules and across the hardware-software interface.

The *structural architecture* consists of subsystems that support execution of operations and connections that represent communication paths. Primitive subsystems represent the basic structural elements, simply called *elements*. Other subsystems may represent hierarchical aggregation. Limiting the taxonomy to the problem at hand, some elements may provide an *execution resource* and/or a *storage resource* (or neither). In general, elements with multiple connections provide an *interface* between these various communication paths. The structural

architecture in Figure 1(b) demonstrates a topology of subsystems that consists of a CPU connected to multiple memories and an I/O controller (IOC). The "CPU" is an aggregation, indicated by the dashed box. Elements such as the "bridge" and "memory controller" serve as interfaces. Elements providing resources are marked with "E" for execution and "S" for storage.

3.2 Functional Representation

Recall that system functionality is described as a set of processes specified with a set of procedures organized as modules. The underlying formalism to express procedure functionality is a control flow graph (CFG) in which nodes are operations and edges are potential flow of control. Operations affect flow of control or modify variables specified as *operands*.

The CFG is a common representation that can be easily derived from an imperative source code specification such as C or Verilog. A formal description of the CFG used in the present technique is found in our previous work [12].

3.3 Resource Mapping

The two views of an architecture are related by a *mapping* that associates operations in the functional architecture with resources in the structural architecture.

Every operation is mapped to an execution resource, and operands, if present, are mapped to a storage resource. Referring to Figure 1, the software modules are mapped to the "core processor" execution resource, and the hardware module is mapped to the "IOC" execution resource (a *fixed* mapping). The mapping of operands to storage resources varies from operation to operation, and in the example of Figure 1, potential storage resources are the core processor registers, DRAM 1, DRAM 2, or IOC registers. Different mapping schemes are considered architecture variations.

3.4 Operation Cost Model

Emphasizing computational simplicity and memory access, a basic model for the average *execution cost* for an operation type, $\bar{c}$, is used. There are three parts: core processor time, instruction fetch time, and an optional memory access time: $\bar{c} = c_{\text{core}} + c_{\text{instr}} + c_{\text{mem}}$, decomposed as:

$$c_{\text{core}} = n\tau_{\text{core}}, \quad c_{\text{instr}} = p_{\text{rd}}^{\text{instr}} m^{\text{instr}} t^{\text{instr}}, \quad \text{and}$$
$$c_{\text{mem}} = p_{\text{r/w}} m_{\text{r/w}}^{\text{mem}} t_{\text{r/w}}^{\text{mem}}$$

where n is core processor cycle count, τ_{core} is core processor cycle time, m is number words accessed and t is the access time from a specific execution resource to a particular storage resource. The superscript of a variable indicates the storage resource, and the subscript indicates a read or write access. Each external memory access component cost exists for an individual operation with some probability, p, which represents a cache miss for a read or the likelihood that a write operation cannot be posted.

The core processor time is associated with an operation through the execution resource mapping, and it includes time to access any elements that execute at the core processor rate, e.g. the time to access processor registers. The cycle count is a databook value and the cycle time is a component parameter.

Operations mapped to a processor execution resource have an instruction fetch cost that accounts for the external memory access time. Note that $m = 0$ for fixed functionality, e.g. for an IOC.

The optional memory access cost, $t_{\text{r/w}}^{\text{mem}}$, is the access time to a particular storage resource from an execution resource and is derived from structural topology by considering the storage resource mapping for each operand. For example, in Figure 1, $t_{\text{r}}^{\text{dram1}}$ is the read access cost to DRAM 1 memory from the core processor.

This abstract model for operation cost neglects numerous details of individual operations, and assumes independence between operations. We anticipate that more sophisticated analysis techniques can be applied to basic blocks to obtain a more accurate execution cost based on micro-architecture details of the target processor.

3.5 Components

A *component* is a set of one or more subsystems and associated connections, i.e. a fixed topology of subsystems, that can be included in an architecture. Components are

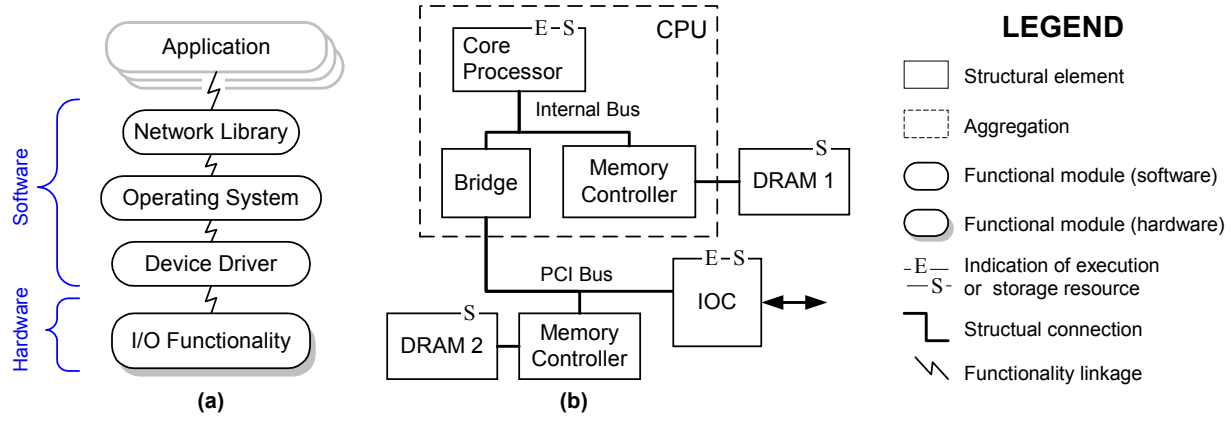

Figure 1. A network interface described by the functional architecture (a) and structural architecture (b).

reusable, may be hardware and/or software, and are generally available from a library. A component may be delivered in many forms, e.g. software source code, an intellectual property core for VLSI design, a synthesizable hardware description language, or a discrete integrated circuit. To clarify explanation, components are assumed to have parameters that can be varied without affecting the fixed topology, e.g. operating frequency or size of memory.

3.6 Architecture Explorations

Our approach assumes that the design space exploration framework delivers candidate architectures for evaluation. The *candidate architecture* is a configuration of components that expresses both a structural architecture, which provides the required physical resources, and a functional architecture that fulfils the system behavior requirements. Candidate architectures are determined by the topology and number of subsystems as well as the population of individual subsystems. The manner in which individual components contribute to an architecture ranges in complexity from a simple subsystem population to instantiating a portion of topology containing several subsystems. Minor architecture variations that do not affect the topology are under control of the designer based on changes to component parameters and the resource mapping.

Figure 2 demonstrates several different candidate architectures. Within the single topology of Figure 2(a), different components can populate a subsystem, e.g. the CPU. Figure 2(b) is a more complex topology that uses an additional physical memory on a separate bus. The potential trade-off is lower cost with a single memory (a), versus improved performance with a second memory "close" to the IOC (b). The impact on system functionality is also demonstrated by the inclusion of different IOCs with the corresponding fixed functionality.

4. Scenario-Based Performance Evaluation

A key contribution of the proposed technique is the application of an evaluation scenario to identify a characteristic subset of functionality that serves as a basis for a performance evaluation. Once a scenario is specified, it is used to generate a pseudo-trace. The operation execution costs are derived from the resource mapping, from which a performance evaluation is calculated.

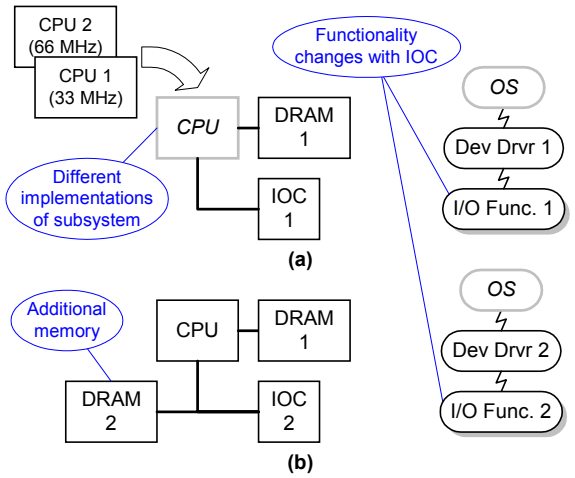

Figure 2. Variations from subsystems or topology.

4.1 Identifying Characteristic Functionality

The functional architecture is incrementally refined into the evaluation scenario through three levels of refinement listed in Table 1. The first step establishes the total set of system functionality, based on the candidate architecture. The second step identifies a subset of the functional specification and the start/end nodes for the scenario. The last step is a detailed analysis of the functional specification that results in an evaluation scenario [13]. The case study in Section 5 quantitatively demonstrates the functional architecture a designer needs to consider in each step.

4.2 Scenario Specification

Intuitively, an evaluation scenario corresponds to a walk through the collection of CFGs that expresses the functional architecture. The evaluation scenario was introduced in the context of software programs in a previous work [13]. Important details are summarized here.

To specify a scenario, the start and end nodes are identified, and each control flow edge is annotated with the number of times it is traversed in the walk. Only control flow operations, i.e. those with multiple outbound edges, need explicit annotation.

User understandable *constraints* are associated with nodes and express dataflow facts, e.g. "a= 0" or "b<2". They are expressed in terms of the functional specification and are interpreted according to the semantics of operations. In this simplified explanation, edge annotations are equivalent to constraining control operation predicates [13].

The flow of scenario specification is shown in Figure 3. At the start of detailed path analysis, the designer has identified a set of CFGs to analyze, and provides the start and end nodes of the scenario. To define a scenario, the

Table 1. Identifying subset of system functionality.

Step What is it?	How formed?	How is designer guided?
1: Functional architecture Set of CFGs	Design exploration framework	Component specifications, resource mapping
2: Identify relevant subset Subset of CFGs, plus start and end nodes	Problem domain identifies interesting behavior	Important system goals, understanding of functional specification.
3: Specify scenario A control flow walk on CFGs	Detailed source code analysis	Scenario specification flow

designer needs to constrain all control operation predicates. An initial pruning cycle analyzes the control dependences between the start and end nodes and extracts predicate constraints that are then applied to the CFG. Based on these constraints, nodes are pruned from the CFG and data flow facts are propagated. A pruned control operation is considered constrained.

If the data flow fact propagation results in new constraints on any control operations, then subsequent automatic cycles continue until no new constraints are found. If there are any control operations left to constrain, a manual constraint cycle begins in which the designer provides a constraint. Using this constraint, more automatic constraint cycles are attempted. When all control operation predicates are constrained, the specification is complete.

Note that key metrics that demonstrate the efficacy of this method are the number of control operation nodes eliminated due to pruning (more is better), the number of constraints found automatically from propagation of data flow facts (more is better), and the number of manual constraints supplied by the designer (fewer is better).

4.3 Pseudo-Trace Generation

The pseudo-trace is generated by the walk from the start node to end node through the collection of CFGs as defined by control flow edge annotations. Each regular operation has only one outbound control flow, and each reachable control operation is constrained to describe how to traverse the edges. Sub-scenarios in a called function can be reused.

4.4 Operation Costs

The variables used in the cost model defined in Sec. 3.4 are determined from component properties, the architecture features, and designer supplied values. Some variables are simply databook values, such as the number of cycles to execute a particular operation (n and τ_{core}).

Variables related to storage resource access are derived from the topology of the structural architecture in combination with databook values. For example, consider the operation "A = M1 + B" from a software component that is bound to the core processor in Figure 1. The variables A and B are bound to register operations, and variable M1 is a memory access bound to storage resource DRAM 1. The register access time is "built in" to the operation execution time. However, the memory access time to DRAM 1 depends on the elements traversed in the structural topology from the execution resource to the storage resource: the CPU internal bus, the CPU memory controller, and the DRAM 1, in this example.

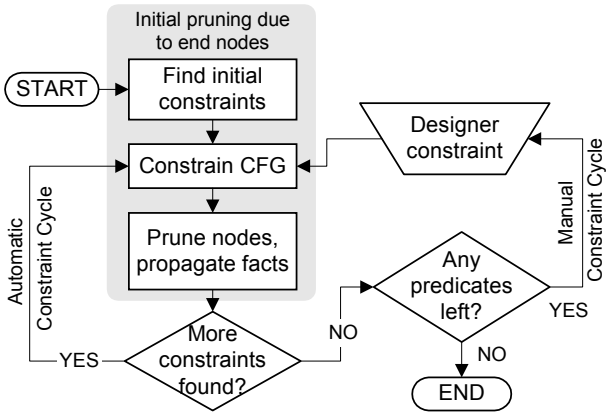

Figure 3. Scenario specification flow.

Given that processor cores typically run an order of magnitude faster than external bus transactions, the configuration of busses and memories external to the processor component tend have a dominant role in an operation cost estimate.

The memory access cost for cached memory regions depends dramatically on the average hit rate. This variable in the cost model is supplied by the designer, an assumption made in several other estimation techniques [1][15].

5. Case Study

The case study applies the proposed technique to a performance evaluation of several candidate architectures based on the system of Figure 1. Two sets of results are reported: the effort to specify a scenario and a demonstration of the fidelity of the performance evaluation across different architectures.

5.1 Experiment Setup

The experiment considers a network interface based on software components from the commercial operating system VxWorks and standard integrated circuits, i.e. hardware components. The CPU component contributes a fixed topology with several subsystems (see Figure 1b). An I/O controller component contributes to both the structural architecture and the functional architecture.

Fixing the functional architecture, the two major structural variations shown in Figure 2 are explored, resulting in four different architectures. First, two different CPU components that differ in core processor speed are used, the Intel i960® RP (1X or 33 MHz core) and RD (2X or 66 MHz core). This illustrates populating a subsystem with different components as shown in Figure 2(a). Second, different topologies of the structural architecture are considered by adding a second storage resource (DRAM 2) shown in Figure 2(b).

5.2 Scenario Specification

The steps of Table 1 are applied to identify a characteristic subset of system functionality.

Step 1. The complex functional architecture is delivered from the exploration framework. The designer specifies the execution resource mapping by associating all software modules to the core processor (the IOC mapping is fixed).

Operands are mapped to the appropriate registers and the DRAM 1 storage resources, under the guidance of a compiler, except that the shared memory between the device driver and "I/O Functionality" is mapped to DRAM 2 when it is present in the structural architecture.

Table 2. Scenario and pseudo-trace summary.

Major Procedure:	Interrupt process	Driver process
Module Size		
Local procedure calls (sub-scenarios)	2	3
Support library calls	4	4
Nodes in local CFGs to be analyzed	177	123
Scenario Specification Summary		
Control ops nodes (to be constrained)	17	13
Automatic constraints found	4	10
Pruned control operations	11	3
Control ops manually specified	2	0
Pseudo-Trace Summary		
Operations in pseudo-trace	196	451
Derived from driver source	46	151
Derived from support library	150	300

Step 2. Beginning with the operating system functionality (723 KB as object code), the contemplated architecture variations lead the designer to focus on the performance of the subset defined by the device driver module (47 KB in size).

The problem domain indicates reception of a small packet is a typical action for this system. The start and end nodes for such a scenario are found using the software documentation and examining the code using a program slicing tool.

Step 3. The detailed analysis applies the algorithm of Figure 3 to identify an evaluation scenario. The analysis examines the device driver source code, packaged as a single file specifying 14 procedures with 1,650 lines of source code. Table 2 summarizes this analysis.

The scenario occurs across two processes: an interrupt routine and the driver executing as a regular task. The reported number of control operations in the CFGs represents how many constraints need to be determined (17 and 13). Note that relatively few operations require designer specified constraints (only 2).

5.3 Performance Evaluation

The number of operations in each pseudo-trace is also listed Table 2. Most operations execute within the core processor in a single cycle of $\tau_{\text{core}} = 30/15$ ns, derived from a databook depending on the 1X/2X CPU. Access times to storage resources are found by analyzing the structural topology, e.g. $t_{\text{r/w}}^{\text{dram1}} = 4\,\tau_{\text{core}}$ and $t_{\text{r}}^{\text{dram2}} = 12\,\tau_{\text{core}}$. The designer supplies effective cache miss rate, e.g. $p_{\text{r}}^{\text{dram1}} = 0.2$.

The performance evaluation of four architectures, one for each scenario process, is summarized in Table 3. These results demonstrate the fidelity of the proposed technique. That is, the relative performance estimates at an architecture-level of abstraction track the observed performance values across the architecture variations. We observed the cost model to be biased towards under estimation, and best suited to explore changes one dimension at a time. The fidelity between the estimates is more important than the accuracy of individual estimates, as this guides selection of promising architectures.

Table 3. Performance evaluation summary. Execution time in microseconds.

Shared mem mapping:		DRAM 1		DRAM 2	
CPU component:		1X	2X	1X	2X
Interrupt Service	Estimate	38	32	38	32
	Observed	34	27	35	27
Receive Task	Estimate	72	55	78	60
	Observed	79	60	98	80

6. Conclusion

A performance evaluation technique was proposed that uses a static model, evaluates for a typical case, and is abstract enough to support early, architecture-level design space exploration. The evaluation scenario approach helps filter a complex set of system functionality to a small, but relevant, subset that serves as the context for performance evaluation.

The case study demonstrates that this technique allows a rapid performance evaluation of candidate architectures that exhibits good fidelity.

7. REFERENCES

[1] W. Ye, et al., Fast timing analysis for hardware-software cosynthesis, *Proc 1993 Int Conf* A. Allara, et al., System-level performance estimation for sw and hw in the SEED project, *Proc ICCD*, 1998, pp 48-53.

[2] V. Basili, B. Boehm, COTS-Based Systems Top 10 List, *Computer*, v 34, n 5, May 2001, pp 91-93

[3] R. Braden, Reqs.. for hosts…, Internet RFC 1122 , Oct 1989

[4] P. Brereton, D. Budgen, Component-based systems: a classification of issues, *Computer*, v 33, n 11, May 2001, pp 54-62

[5] L. Friedrich, et al., A survey of configurable component-based operating systems.. , *IEEE Micro*, v21, n3, May-June 2001

[6] T. Gupta, et al., Processor evaluation in an embedded systems design environment, *Proc. Conf VLSI Design*, 2000

[7] J. Jahnke, Engineering component-based net-centric sys…, Proc. ESEC, *ACM SIGSOFT Eng Notes*, v 26, n 5, Sept 2001

[8] K. Lahiri, A. Raghunathan, S. Dey, Fast performance analysis of bus-based .. , *ICCAD*, 1999, pp 566-572.

[9] Y. Li, S. Malik, A. Wolfe, Performance estimation of embedded software .., *Trans Des Auto Elec. Sys*, v4 n3, Jul, 1999

[10] J. Liu, et al., Software timing analysis using HW/SW cosimulation and instr. set simulator, *CODES*, 1998, pp 65-69.

[11] F. Moya, J. Moya, J. Lopez, Evaluation of Design Space Exploration Strategies, *Proc 25th EUROMICRO Conf*, 1999

[12] J. Russell, Program Slicing for Codesign, *CODES*, 2002.

[13] J. Russell, M. Jacome, Scenario-Based Software Char. as a Contingency to Traditional Program Profiling, *CASES 2002*

[14] M. Sparling, Lessons learned through six years of component-based development, *Comm. ACM*, v43, n10, Oct 2000

[15] K. Suzuki, A. Sangiovanni-Vincentelli, Efficient Software Performance Estimation Methods …, *DAC 1996*

[16] B. Tabbara, et al., Fast hardware-software co-simulation using VHDL models, *DATE*, 1999, pp 309-316.

[17] *Comp. Design*, pp 452-457