Timing Driven Co-design of Networked Embedded Systems

Dinesh Ramanathan Ravindra Jejurikar Rajesh K. Gupta
Center For Embedded Computer Systems
Department of Information and Computer Science
University of California
Irvine, California 92697.
{dinesh,jezz,rgupta}@ics.uci.edu

Abstract

Advances in microelectronics integration have led to emergence of tightly integrated systems with high performance network interfaces. Design of such systems especially for single chip implementation is a delicate balance of functionality and available time budget to perform the tasks. Computer-aided design tools and methodologies are needed to ensure the correctness of the design and efficiency of the design process, especially for networked systems that have strict timing requirements both due to technology as well as networking needs. We present an overview of a timing-driven design methodology for networked systems, developed at the University of California, Irvine.

I. Introduction

Networked embedded systems (NES) are distributed embedded systems connected together using network interfaces and standardized protocols. The protocols are often implemented partly in hardware and partly in software. NES are usually designed at the highest level of abstraction as a collection of basic network models communicating with each other. Typically, this communication is performed under strict timing constraints and based on some standardized protocol. At the highest level of abstraction, these systems are modeled in an environment that enables their simulation. Many simulation environments at this level provide support for commonly used network models, for instance the network simulator NS [13] supports models for TCP, randomized input generation, multicast transport, etc.

A high-level description of a networked embedded system is essentially a graph whose structure corresponds to the architecture of the system. The nodes of the graph represent subsystems and the edges the communications between them. Some NES nodes are implemented as an application specific integrated circuit (ASIC), or a combination of hardware and software that must deliver its performance in real-time and in a very time constrained environment. Timing constraints in such systems arise from a variety of sources: (a) hardware related; (b) reactive environment - interfaces to sensors etc; (c) network quality of service requirements. These constraints arise from the requirement of integrating the complete design on a single chip as well as ensuring that its functionality meets the required network protocol.

The problem is that many timing constraints depend upon network system performance and expectations. The network protocol has to route packets of information from the inputs to the outputs within a specified time interval giving rise to end-to-end timing constraints. The implementation of each NES node must, therefore, be done in the context of the network topology, requirements and constraints in which these are embedded. This process is iterative and designer driven. In this paper, we outline a systematic methodology that allows the designer to construct and use network-level simulation models in the actual implementation (constraints) of the individual NES nodes. This consists of (a) an ability to capture timing constraints for the NES nodes by using network-level simulations and (b) capturing the effects of the network topology in NES implementation as well.

II. Timing in Co-Design

Though timing plays an important role in the design of networked embedded systems, it usually becomes a design consideration only at the hardware design stage (register transfer level). Further, the problem of designing a temporally correct networked embedded system is a difficult one [1, 5, 20, 4, 6, 9, 10]. In practice [20], the emphasis is usually on designing a functionally correct system [21]. The temporal correctness of the system is usually verified after the system's components are integrated and the resulting system's functional correctness is ensured. This approach usually results in expensive re-design iterations in order to satisfy temporal constraints for both hardware and software subsystems.

Timing models for networked embedded systems are often constructed at multiple levels. For instance, a network model would focus on the system performance as a node in a distributed system; a hardware interface model

may focus on system interface timings, etc. These are, of course, related. A highly constrained network interface has implications for what is implemented in hardware or software. Similarly, underlying hardware timing performance determines network buffering and interface protocol needs. Timing requirements arise at various levels of abstraction as well. At one level, the system level timing is paramount. At this level, the only consideration is the end-to-end constraint of the system. At a level where the design has been refined to hardware and software, there could be end-to-end constraints for hardware and software as well as throughput constraints for both hardware and software.

We believe that not only is it possible to specify, explore and exploit the timing information in high-level network/process-centric models, but also that such timing information can be used to derive time budgets to reduce the total design cycle time. By having timing information about the various subsystems of the design, system designers can make better hardware/software partitioning decisions. In addition, system designers can provide both hardware designers and software designers timing bounds which must be met in order for the system to meet its timing requirements. As a result, timing requirements trickle down from one level of abstraction to another as the design gets refined. Such an early decomposition of timing performance between hardware and software helps reduce design iterations, design time and enables timing driven co-design.

III. Modeling Network Embedded Systems

Simulation allows the evaluation of network protocols under varying network conditions. Studying such protocols before implementing them is critical to exploration of design alternatives. Our approach unifies analysis of networked embedded systems that rely upon highly detailed timing models from network-centric models. This methodology relies on techniques for rate derivation and the use of network simulators such as NS [13] developed by the VINT project at the University of Southern California and the University of California, Berkeley.

Generation of high level timing models using rate derivation has been studied in detail in [17, 18, 19] and implemented at the University of California, Irvine in the framework called RADHA-RATAN that works on a generalized task graph. A *generalized task graph* [17] is a collection of nodes and edges. The nodes represent functionality and the edges transport tokens (pieces of information, both data and control). Nodes of the task graph are classified based on their behavior when they receives tokens from their predecessor nodes. RADHA-RATAN is a collection of algorithms that generate time budgets for the nodes of the task graph based on the rate at which the nodes exchange tokens.

NS supports a variety of network protocols and modeled functionalities. For instance, TCP behavior – where the focus is on selective, forward and explicit acknowledgments; router queuing – where the focus is on random early detection, congestion notification etc., multicast transport; and multimedia – where the focus is on layered video, quality of service and transcoding [11]. This allows us to build a complete simulation model of the system in which NES are used. We discuss our results by integrating NS [13, 11] with the timing analysis tool RADHA-RATAN, to generate timing models [8] for the networked embedded systems.

A. Overall Design Flow

Networked embedded systems are conceptualized at the highest levels by focusing on functionalities such as congestion, routing, quality of service, etc. The simulators are used to analyze requirements such as the queue length, rate of growth, etc. The goal of such simulators is to determine whether the conceptualized system meets throughput requirements, quality of service requirements and adhere to standardized protocols (which typically specify throughput and allowable error rates). Note that the designer is not yet concerned with the implementation of subsystems. As a result, network simulators work at a very high level of abstraction. The typical starting point in the design of a networked embedded system using NS is a graph called the *network graph* where the nodes represent subsystems and the edges represent communication channels between these subsystems. The nodes of the network graph exchange tokens among themselves to simulate packets or pieces of information that must be exchanged between them in order to collectively perform the task of the system. At this level, NS allows the designer to specify distributions (for example, uniform or normal) or protocols (for example, TCP) from which the input to the system is generated. This constitutes the environment whose stimuli the system reacts to. Under these input considerations, the designer observes the performance of the system in terms of the tokens passed between the various nodes of the graph. Note that at this level, the system designer does not know the size of a token nor have a classification for the nodes of a graph in terms of their behavior. As a result, the designer, based on previous engineering experience, attempts to mutate the NS graph to generate a high-level model of the system that will meet the system requirements at this level of abstraction.

Consider a specific portion of the system. This portion can be represented as a single node of the network graph or a collection of nodes that implement a specific functionality. The designer refines the subsystems (node or a subgraph of the network graph) to capture its functionality and perform tradeoffs for the refined subsystem. Further, after this refinement phase is complete, the designer should be able to run a system level simulation in NS to validate the design choices for the subsystem under consideration. Note that while performing this refinement, it

is not possible for the designer to obtain a timing model for the subsystem.

Further, after a network graph – that models the system and is likely to meet the timing requirements – is obtained, the designer targets specific portions of the system, refines the design by specifying the width of the tokens and the behavior of each subsystem. The implementation details of any individual NS node is not known at this level. Since timing information cannot be represented within NS, designers use ad hoc techniques to perform tradeoffs between requirements and the performance of the system.

B. A Two Tier Timing Model

The goal of mutating (or structuring) the initial network graph is to obtain a network graph that will meet the system's timing requirements. Further, while refining a specific subsystem the designer is performing a similar task. However, in both cases, the designer determines the structure of this graph and the rate at which they pass tokens around i.e. communicate with each other. Specifying the rate at which the input produces tokens is the only control the designer has when attempting to refine the functionality of a subsystem. The rate at which tokens are produced by internal nodes of the network graph (nodes that do not interact with the environment) are being determined by the designer in an ad-hoc manner. Further, determining the size of the queues needed within each subsystem (node of the graph) is essentially determining how much slower that particular subsystem must function as compared to the nodes that send it data in the form of tokens. In general, determining the queue length is determining the rate at which a particular node should operate so that it does not loose tokens but still meet its error rate specifications.

There is no formal model to account for timing within the realm of NS and other network simulators. As a result, there is a disconnect between the NS models and implementation models that could be derived from NS performance models. To address this problem we use a two level timing model which consists of a network graph and a generalized task graph model. We define a two level model for timing for the network graph and the subsystems represented by the nodes of the network graph.

The first is at the NS level and is built from the network graph. This process is called timing driven task structuring. It involves structuring the network graph so that the task graph generated from the NS level meets the system's timing requirements. The task graph generated from the NS representation captures the timing behavior of the networked system. The designers define the environment of the system and mutate the structure of the network graph (that represents the system) until they are satisfied with its timing behavior. At this stage, the designers also decide on a hardware/software partition. This partitioning is also timing driven. In essence, task structuring provides the link between requirement analysis and system design

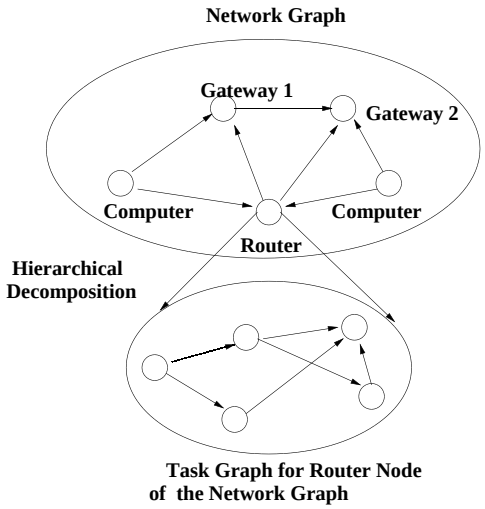

Fig. 1. *Hierarchical Decomposition of Network Level Nodes into Generalized Task Graph*

as shown in Figure 2 and is applied to the network graph to automatically extract a generalized task graph from it. If the network graph is simple this process is automated, if not, we merely capture the input and output requirements for each node of the network graph. We provide a set of attributes that can be annotated onto an NS node which allows us to extract timing information from an NS node using a formal timing analysis technique that models timing within the NS domain. The combination of NS and a formal timing analysis technique provides a methodology by which the designer can determine the rate at which the nodes of the NS network graph created must work in order to meet their timing requirements. Section IV describes how hardware/software partitioning can be timing driven and performed on the network graph.

The second level is at the refinement of a particular subsystem. At this level, we propose the construction of a generalized task graph [16] within NS. This graph is constructed using the basic node components library we have provided within NS. Once the generalized task graph has been constructed, we can generate time budgets for the nodes using the rate analysis tool called RADHA-RATAN [16]. Now, the designers can determine the rate at which the internal nodes in the generalized graph produce tokens based on the behavior that the designer prescribes for them. The designers also have control over the rate at which the input nodes of the network graph produce tokens for consumption by various subsystems. At this level, we refine the functionality of each node (subsystem) in the network graph using a generalized task graph in its implementation environment. This allows the designer to generate a timing aware model for each subsystem of the networked system. We refer the reader to [16, 17, 18] where this approach is described and explained in detail.

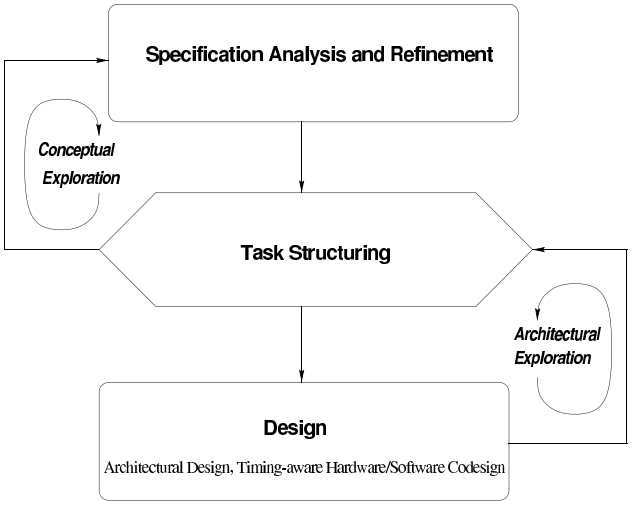

Fig. 2. *Task structuring provides the link between requirements analysis and system design*

C. Timing Driven Structuring of the Network Graph

Task structuring [24] captures the parallelism inherent in the design of the system. The design of the network task graph allows the designer to structure the subsystems in ways that exploit parallelism and pipelining. Parallel tasks allows the designer to tradeoff resources and timing performance, and pipelining allows the designer to tradeoff throughput and latency.

In order to specify timing using the network graph, we must generate a graph from it that is timing aware. This is done by converting the network graph into a directed graph called a generalized task graph [16, 17, 18] (the task graph for short) and generating timing information for it based on the communication protocol of the tokens exchanged among the nodes of the task graph. This procedure is described in detail in [8]. The designer structures the network graph to satisfy the performance requirements of the system. For each network graph structure, we generate a task graph and iteratively apply RADHA-RATAN to the task graph. RADHA-RATAN generated the timing budgets for the internal nodes of the task graph. This enables the designer to ensure that the system meets its timing requirements. The iterative procedure of generating a task graph from a network graph and extracting timing information from it continues till the designer is satisfied with the structure of a network graph that meets timing requirements.

The task graph [17, 16, 23, 18, 19] corresponds to the data/control flow diagram of a subsystem that implements a node in the network graph. Each node represents a task, and each edge represents a unidirectional communication channel between its *producer* and *consumer*. Both data and control flow through a channel are modeled using token flow. The granularity of every token is *channel-specific*, and once fixed, the tokens are indistinguishable within and across channels for timing analysis purposes. We relax this assumption to model communication between various subtasks of the system.

The sensors and actuators of the system are also included in its task graph, in which they are referred to as *input* and *output tasks*, respectively. We assume that each task is periodic or sporadic, i.e., each task has a bounded rate interval. The *rate* of a task is equal to the number of its executions per unit time. The *period* of a task is equal to the time it takes for one execution. The difference between the task graph and the network graph is the level of abstraction they use. If the network graph is constructed at a very high level, it cannot be converted into a generalized task graph automatically. Typically, network graphs contain statistical performance characterization and are designed at higher levels of abstraction which sometimes obscures implementation details. However, we believe that all network graphs can be hierarchically decomposed into task graphs [8], but we leave that as an open problem and do not discuss it here.

Recall that a task graph is constructed in one of two ways: it is extracted from the network graph or explicitly constructed to represent a subsystem. Once a task graph is available, the tool RADHA-RATAN which contains derivation and validation algorithms for both rate and separation constraints can be applied to it. This would provide the designer with the rates at which the tasks of the task graph would have to work at in order to meet the system's (or subsystem's) timing requirements. The algorithms in RADHA-RATAN, developed at UC Irvine are explained in detail in [17, 16, 23, 18, 19]. RADHA-RATAN requires that only the rates on the inputs to the system be known. It derives the rates on all internal tasks. Based on these derived rates, it can also validate rates that the user may have imposed on internal tasks. The derived rates depend only on the rates of the inputs and the task graph. It does not require the internal details of each task.

D. Modeling Communications

While designing the network graph, designers have to deal with modeling communication between various subsystems. Communication between the various subsystems can impact system performance. If communication can be modeled at a higher level of abstraction, communication between various subsystems can be considered to be another function of the system and can be modeled within the task graph model.

Initially, we generate time budgets on tasks with the assumption that the tokens (data and control flows) are *user specific* and *channel specific*. This implies that different channels can have different kinds of tokens being passed between tasks. Channels transport the same token, but the semantics of tokens could differ across channels. Initially, we also assumed that once the tokens were fixed, they were indistinguishable within and across channels and token delivery time was a part of the time budget for a particular task.

Communication is modeled using channel nodes [18] in the task graph along with canonical tokens that are user and channel specific. In order to model interfaces between various tasks, we explicitly expose the token delivery delay: we use *channel nodes* in the generalized task graph to represent channels with delays. This extension to the task graph model is powerful enough to handle the sharing of these communication channels among various tasks. To model sharing of communication channels, we define an equivalence relation on edges of the task graph. This equivalence relation is user specific. Two channels may lie in the same equivalence class because they transport a similar number of tokens or they share a similar implementation or for any other reason the user deems fit.

In addition, in [18] we introduce the notion of a canonical token using which we can assist the system designer in picking a suitable implementation for a channel. This issue has great impact on the design of the system since tokens and channels bind the entire system together. Using this methodology, we can model complex communication schemes in embedded systems at a *very high-level*, where communication can range from bit transfers to transfer of extensive (and voluminous) data sets these are modeled uniformly and incorporated in to the timing analysis.

Now, our system level task graph is at the stage where we have modeled the communication between tasks and allowed sharing of communication channels within the generalized task graph model. Using this high-level representation and abstraction, we have characterized the communication channels and can analyze their timing requirements. In the partitioning stage, we can now select a specific implementation for the communication channels that would meet the timing requirements of the system.

IV. Task Refinement and Process Timing Simulation

Once the network graph has been through timing driven task structuring [24] and the designer is satisfied with its timing performance, we move to partitioning the system into hardware and software components. At this stage, we can choose to attribute each node of the task graph, or a collection of nodes of the task graph to hardware or software. After this partitioning has been done, each subtask of the system level task graph is hierarchically designed in its chosen environment (hardware or software) using essentially the techniques illustrated so far. Figure 1 shows a system level task graph and a task graph that represents a single node of the network graph.

The timing budgets derived for each subtask now become the end-to-end constraints on the design of each subtask. In this way, each subtask now becomes the system that needs to be designed within an environment described by the whole system. Each subtask is now designed within the task graph framework using the the same procedures described here, i.e. timing driven (subsystem) task structuring using RADHA-RATAN and simulation. If the interface of the subsystem is clearly defined while defining the network task graph, each subsystem can be seamlessly integrated into the network graph. Further, the generalized task graph can be transformed to a model which supports event based simulation semantics. This is done by automatically generating Verilog code from the task graph model as described in [24]. Also, in [8] we have shown that the task graph can be modeled within NS to support event based simulation semantics. This enables system level simulation throughout the design cycle of the system.

The simulation is performed using the notion of *process timing simulation* of the entire system that consists of timing model of NES without resorting to simulation of its detailed functionality. Process timing simulation is an abstract simulation model that captures the token exchange protocol as well as the rate at which the tokens are exchanged (obtained from RADHA-RATAN) between various nodes of the network graph and the generalized task graphs that represent each node of the network graph. In this simulation, we ignore the specific functionality of each generalized task graph node, and only model the token exchanged and the rate of production and consumption of tokens within the entire system. Process timing simulation of the entire system enables a designer to fine-tune the timing requirements of the system, especially those of the nodes representing channels, thereby obtaining better bounds on the timing parameters of the system. Process timing simulation has been implemented within the NS framework as shown in [8].

Now, while designing each subsystem downstream in the design process, if the subsystem designer concludes that some of the required timing constraints cannot be met, the system designer can make decisions on the design of the complete system based on this information. As a result, timing is accounted for during the design of the entire system.

We have applied this methodology to design an ATM connectionless server [14, 15, 8] which had strict timing constraints: an ATM packet had to be routed within the the server in 125ms. Using the described methodology, we made timing driven decisions at higher levels of abstraction which allowed us to very quickly converge to a network graph that met the timing requirements. Our approach enabled partitioning decisions to be timing driven as well. Further, each subsystem of the server was designed using a task graph to model the behavior of the system within its environment. We compared our timing driven methodology against a conventional design methodology. Most of the timing decisions reached by designers using the conventional methodology were reached by us in a much shorter time and in a systematic manner which greatly reduced the overall design time of the system.

V. Summary and Conclusions

Efficient design of time-responsive networked systems requires that the design process be driven at the highest level of abstraction by timing and requirement analysis. The interplay between timing and non-timing requirements at the system level can be used to make effective design tradeoffs (in cost, performance) and reduce design time. However, extensive reliance on application-specific design models and methods delays exploration of detailed timing specifications until the detailed design phase, thus limiting the scope of any architectural-level design explorations. The need is for modeling methods and tools that help the system designer drive the transition from requirement analysis to design using timing specifications.

Our strategy is to carry out timing analysis at higher levels of abstraction as well where the number of problem parameters is small, and to propagate the results obtained at one level to another throughout the design cycle. When there is uncertainty that makes it impossible to do analytical analysis, or there is a need to enhance the design process, we resort to iterative simulation [3]. To this end, we think that being timing-driven has three important parts: analysis, simulation and design interaction. The RADHA-RATAN framework developed at UC Irvine covers the analysis part [2, 17]. The NS simulator developed by the VINT [13] project covers the simulation part. An important aspect of this work is that it treats timing abstraction orthogonal to abstraction of the functionality. In other words, the timing model can be fairly detailed (for instance, the generalized task graph model in [17]) while keeping the functionality at a very high-level of description barely enough to carry out task-level system structuring. This provides for a consistent model transfer between an NS simulation and RADHA-RATAN which annotates the simulation with timing information.

Acknowledgments

The RADHA-RATAN framework has been developed by Ali Dasdan, Anmol Mathur and Dinesh Ramanathan. The authors would like to acknowledge support from National Science Foundation award numbers MIP 95-01615 (CAREER) and CCR-9806898, and from DARPA DABT63-98-C-0045.

References

[1] Amon, T. Specification, simulation, and verification of timing behavior. PhD thesis, Univ. of Washington, 1993.

[2] Dasdan, A., Mathur, A., and Gupta, R. K. RATAN: A tool for rate analysis and rate constraint debugging for embedded systems. In *Proc. European Design and Test Conf.* (1997), IEEE, pp. 2–6.

[3] Dasdan, A., Ramanathan, D., and Gupta, R. K. An interactive validation methodology for embedded systems. In *Proc. High-Level Design Validation and Test Workshop* (Nov. 1997), IEEE, pp. 123–30.

[4] Hulgaard, H., Burns, S. M., Amon, T., and Borriello, G. An algorithm for exact bounds on the time seperation of events in concurrent systems. *IEEE Trans. Comput. 44*, 11 (Nov. 1995), 1306–17.

[5] J. E. Coolahan, J., and Roussopoulos, N. Timing requirements for time-driven systems using augmented petri nets. *IEEE Trans. Software Eng. 9*, 5 (Sept. 1983), 603–16.

[6] Jahanian, F., and Mok, A. K.-L. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng. 12*, 9 (Sept. 1986), 890–904.

[7] McMillan, K. L., and Dill, D. L. Algorithms for interface timing verification. In *Proc. Int. Conf. on Computer Design* (1992), IEEE, pp. 48–51.

[8] Ramanathan, D., Jejurikar, R., and Gupta, R. K. Timing modeling and analysis for networked embedded systems. Technical report, University of California at Irvine, Nov. 1999.

[9] Xu, J., and Parnas, D. L. On satisfying timing constraints in hard real-time systems. *IEEE Trans. Software Eng. 19*, 1 (Jan. 1993), 70–84.

[10] Yen, T.-Y. Hardware-software co-synthesis of distributed embedded systems. PhD thesis, Princeton Univ., 1996.

[11] http://www–mash.cs.berkeley.edu/ns/ns-research.html

[12] Balarin, F., et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publ., Boston, MA, USA, 1997.

[13] Sandeep Bajaj et. al. Improving Simulation for Network Research University of Southern California Technical Report 99-702, March 1999

[14] D.P. Hong, B.J. Vickers and T. Suda, "Connectionless Data Server for Public ATM Networks: Design and Perfo rmance" TR #94-10, Department of Information and Computer Science, Universit y of California, Irvine, 1994.

[15] D.P. Hong, B.J. Vickers, T. Suda and C. Oliveira "The Internetworking of Connectionless Data Networks over Public ATM: Connectionless Server Design and Performance" TR #94-41, Department of Information and Computer Science, Universit y of California, Irvine, 1994.

[16] Dasdan, A., Ramanathan, D., and Gupta, R. K. A timing-driven design and validation methodology for embedded real-time systems. *ACM Trans. on Design Automation of Electronic Systems. 3*, 4 (Oct. 1998).

[17] Dasdan, A., Ramanathan, D., and Gupta, R. K. Rate derivation and its applications to reactive, real-time embedded systems. In *Proc. the 35th Design Automation Conf.* (1998), pp. 263–268.

[18] Ramanathan, D., Dasdan, A., and Gupta, R. K. High-Level Modeling of Communication in Real-Time Embedded Systems. *IEEE High-Level Design Validation and Test Workshop*, Nov 1998, pp. 172–180.

[19] Dasdan, A. Timing Analysis of Embedded Real-Time Systems. Ph.D Thesis, University of Illinois at Urbana-Champaign, 1998.

[20] Gerber, R., Hong, S., and Saksena, M. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Software Eng. 21*, 7 (July 1995), 579–92.

[21] Gomaa, H. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley, Reading, MA, USA, 1993.

[22] Hatley, D. J., and Pirbhai, I. A. *Strategies for Real-Time System Specification.* Dorset House, New York, NY, USA, 1987.

[23] Mathur, A., Dasdan, A., and Gupta, R. K. Rate analysis of embedded systems. *ACM Trans. on Design Automation of Electronic Systems 3*, 3 (July 1998).

[24] D. Ramanathan and A. Dasdan and R. K. Gupta Timing Driven HW/SW Codesign Based on Task Structuring and Process Timing Simulation IEEE Workshop on Hardware Software Co-design (CODES), May 1999