An Energy-aware Synthesis Methodology for OS-driven Multi-process Embedded Software

T. K. Tan†, A. Raghunathan‡, and N. K. Jha†
† Dept. of Electrical Eng., Princeton University, NJ 08544
‡ NEC America Labs, Princeton, NJ 08540

Abstract

The growing software content in various battery-driven embedded systems has led to significant interest in technologies for energy-efficient embedded software. While low-energy software research has, in the past, focused on energy optimization at the instruction and source-code levels, approaches targeted at a higher software level are beginning to gain attention.

In this work, we propose a methodology to refine a control-data flow diagram (CDFD) model of an embedded software program into an energy-efficient multi-process software architecture graph (SAG). Our starting representation, the CDFD, is capable of modeling data-dependent control flow. Energy efficiency is achieved by reducing the energy wastage due to context switches and inter-process communications (IPCs). Conditionally-unused computational operations, and their corresponding energy, are also avoided by a condition-aware static scheduler. Finally, code generation is performed from the energy-efficient SAG to produce a multi-process program that implements the original CDFD specification. Experimental results establish the efficacy of the proposed approach.

1 Introduction

The conventional approach to energy optimization of embedded software has been to adapt various stages of the software compilation process to model and minimize energy consumption. A complementary body of work attempts to best utilize power saving features, such as shutdown based power management and dynamic voltage scaling, that are available in modern embedded processors. Both of these approaches have been widely researched. Energy optimization at the software architecture level is an emerging approach that is complementary to conventional low-power software techniques, while promising significant energy savings. In this paper, we model an OS-driven multi-process embedded software at the *behavior level* using the CDFD. We also present a comprehensive software synthesis methodology that synthesizes an energy-efficient SAG from the CDFD, and performs static scheduling and code generation for the synthesized SAG. The proposed CDFD is capable of modeling data-dependent control flow behavior. Our condition-aware static scheduling approach also saves energy by avoiding conditionally-unused computational operations.

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-02-C-P302.

Related work

Low-energy software research has gained momentum in the last decade. At the instruction level, the ideas generally center around low-energy adaptation of one or more steps in the compilation process, including transformations, instruction selection, register assignment, instruction scheduling, *etc.* [1, 2]. At one step above the instruction level, energy impact of source code transformations has been studied in [3, 4]. The synthesis of low-energy software architecture, and the corresponding program code, starting from a concurrent behavioral representation, however, has not been investigated so far. Therefore, we focus on this area in this work.

Multi-threaded software synthesis targeted towards a dynamically scheduled environment has been investigated in [5]. It uses a *software synthesis script* that translates a specification based on a *constraint graph* into a set of threads, and performs static scheduling within the threads. In the reactive system domain, efficient compilers have been built for different programming languages based on the synchronous/reactive model of computation. Software synthesis for reactive systems has also been investigated in [6] as part of the hardware-software co-design tool called POLIS. In another regime of software synthesis, compile-time scheduling of Petri-net models have been proposed in [7]. Software synthesis based on free-choice Petri nets, an extension of Petri nets to include data-dependent control, has been investigated in [8, 9].

The rest of the paper is structured as follows. In Section 2, we discuss the motivation for our work. This is followed by a detailed presentation of our energy-aware software synthesis methodology in Section 3. Section 4 presents the experimental results. In Section 5, we give the conclusions.

2 Motivation

In this section, we address a fundamental question that leads to the motivation of our work. The question is: how does energy-aware software architecture synthesis help produce a multi-process software program with an energy-efficient software architecture?

Consider a behavioral specification of a software subsystem as shown in Fig. 1. The circles represent the *actors*, or the data transformation functions. As in [10] or any other data flow model, an actor fires its outputs when all its inputs are available. The empty rectangular boxes in Fig. 1 represent passive devices, which are devices that produce data tokens on request. The time given next to a device is its response time. The response time of a passive device is the time the device requires to produce the data after a request is

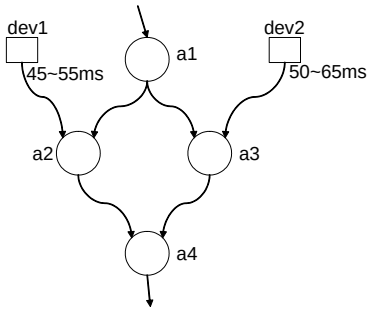

Figure 1. Behavioral specification for a software sub-system

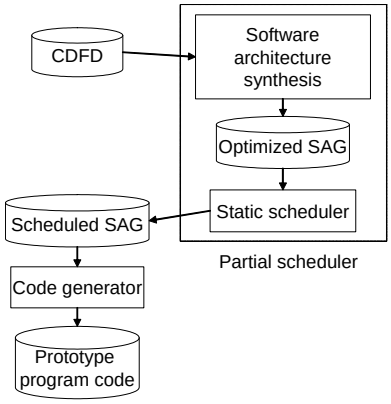

Figure 2. The flow diagram for the overall synthesis methodology

made. For example, *dev1* in Fig. 1 has a response time ranging from $45ms$ to $55ms$. Furthermore, we assume that the computation delay of the actors is insignificant compared to the device response time.

Given that there are slow passive devices in the behavioral specification, we should not generate a single-process implementation of this sub-system because doing so will unnecessarily delay the activation of actor $a4$. Instead, we require at least two software processes so that actors $a2$ and $a3$ are mapped to separate software processes. The other actors ($a1$ and $a4$) can be mapped to either process. Some combinations are: (1) $(a1, a2, a4; a3)$, (2) $(a1, a2; a3, a4)$, (3) $(a1, a3; a2, a4)$, *etc.*, where, for example, combination (1) implies that $a1$, $a2$ and $a4$ are in one process, and $a3$ in another. Clearly, if different amounts of data are passed along different edges, some of the above combinations will be more energy-efficient than others. A software architecture synthesis methodology is *energy-aware* when it attempts to look for the most energy-efficient combination while exploring all the combinations that are *valid*. A valid combination, like those listed above, does not incur unnecessary delay.

3 A Methodology for Energy-aware Software Synthesis

In this section, we describe our methodology in detail.

A flow diagram for the overall software synthesis methodology is shown in Fig. 2. The objective of this methodology is to generate a multi-process software program with an energy-efficient software architecture. The

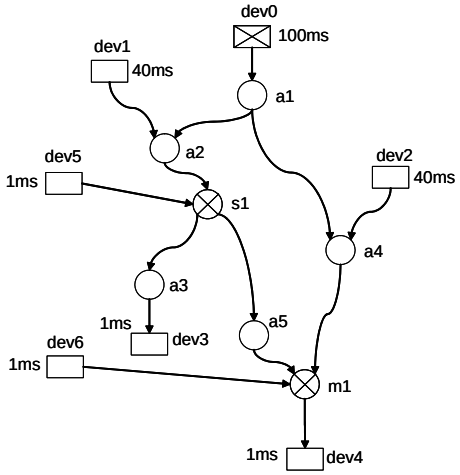

Figure 3. A CDFD representation of an embedded software specification

key components of the methodology include: (1) a concurrent behavioral representation of the embedded software, (2) a representation for the scheduled multi-process software architecture that is readily convertible to program code, (3) an approach to convert the behavioral representation into the multi-process software architecture, and (4) a program code generation step. These key components are described separately in the following subsections.

3.1 The CDFD model

The CDFD model is a hybrid of the data flow diagram (DFD) [11] and token flow models [10]. An example of the CDFD representation of an embedded software is shown in Fig. 3. Five types of entities are featured in the CDFD: (i) active devices, (ii) passive devices, (iii) computation actors, (iv) split actors, and (v) merge actors.

An active device, denoted by a crossed rectangular box, initiates data transfer spontaneously. An example of an active device is a timer. For the purpose of considering only the average behavior, we use an average activation period to model data token generation of an active device. Passive devices are denoted by empty rectangular boxes. They do not initiate data transfer. Instead, they wait for the actors (computation, split or merge) to initiate data transfer. When an actor requests a data token from a passive device, the *response time* of the passive device is the time it takes to produce the data token.

The computation actors in the CDFD are denoted by circles in Fig. 3. Basically, once scheduled, a computation actor in the CDFD makes a data token request to all its inputs, and waits for the availability of all its input data tokens before it fires its outputs. For example, when actor $a4$ in Fig. 3 is scheduled, it makes data token requests to actor $a1$ and device $dev2$, waits for the data token to be available, and fires. Once it fires, its output token is available to actor $m1$.

The split and merge actors are both denoted by circles with cross-bars in Fig. 3. A split actor has a Boolean input from the side, and a normal input from the top. It routes the input to the left output if the Boolean input is true, and to the right output if the Boolean input is false, *e.g.*, actor $s1$. A merge actor also has a Boolean input from the side, but two normal inputs from the top. It routes either the left or the right input to the output based on the value of the Boolean input, *e.g.*, actor $m1$.

3.2 The SAG model

There are two types of entities in an SAG: components and connectors. The notion of components is further refined to consist of two types: computation and device components. Each computation component hosts a set of statically scheduled (totally ordered) actors. A device component, on the other hand, only hosts a single device, active or passive. Connectors serve as the "glue" parts that connect the components together. There are also two types of connectors: communication and device-IO. A communication connector links two computation components, whereas a device-IO connector links a device component to a computation component.

The SAG model as a whole has been conceptualized to emulate the operation of an OS-driven multi-process embedded software. For example, the computation components are synthesized into software processes in an OS-driven run-time environment, the communication connectors are mapped to the OS-supported IPC channels, and the device-IO connectors are mapped to the OS-supported device drivers. The device components, in the same manner, can be mapped to device emulation code if needed. Since the program code for the connectors is part of the OS in an OS-driven software, the program code generated for the computation components interfaces with the connectors by making system calls.

The operation of the computation components, or software processes, is modeled by two complementary operational models. One is an extended directed acyclic graph (exDAG), the other is a state diagram. The nodes in the exDAG have the same behavior as the actors in the CDFD. These are computation nodes, split nodes, and merge nodes. However, no device is defined within the exDAG since devices are modeled within the device components. *IN* and *OUT* nodes are defined within the exDAG as the interface between the exDAG nodes and other entities. The state diagram in a computation component, on the other hand, is a static schedule of the operation flow represented by the exDAG. Static scheduling of an exDAG into a state diagram can be performed based on a topological sort of the exDAG. This static scheduling algorithm is presented in Section 3.4.

3.3 The partial scheduler

The process of converting a CDFD into an SAG is called *partial scheduling* because an SAG is essentially a partially ordered CDFD in the sense that the actors within a software process are statically (totally) ordered whereas the ordering between actors from different software processes is not determined until run-time.

Scheduling (partially) a CDFD consists of two major steps. The *first* step, called software architecture synthesis, divides the CDFD into a set of non-overlapping subgraphs, each of which is separately mapped to a software process. The *second* step performs static scheduling within each process. This two-step procedure is illustrated in Fig. 4 using the CDFD example from Fig. 3. The input to the software architecture synthesis step is a CDFD shown on the left. The output from the software architecture synthesis step is an SAG without the state diagram, as shown on the right of Fig. 4. Within each process of an SAG is an exDAG, which is a subgraph of the original CDFD. In Fig. 4, the processes are labeled $proc1$ to $proc3$, whereas the connectors (communication or device-IO) are numbered $j1$ to $j10$. Edges between processes and connectors indicate the utilization relationship. For example, process $proc2$ is connected to connector $j6$ by an edge since it utilizes $j6$ to pass data to process $proc3$. The devices are kept intact, with the same graphical notations as those in the CDFD. After the software architecture synthesis step, the SAG is subjected to

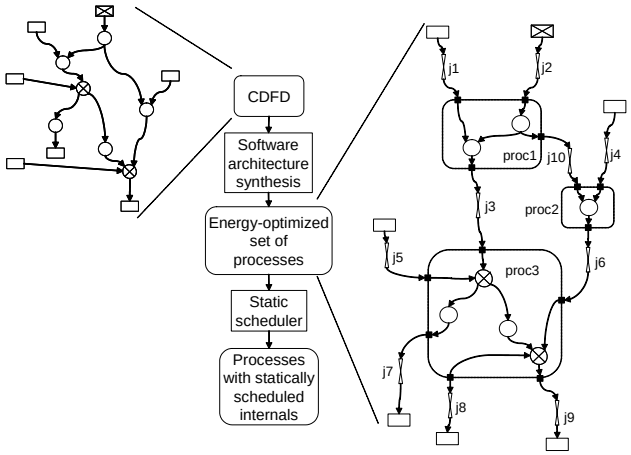

Figure 4. The two-step partial scheduling procedure

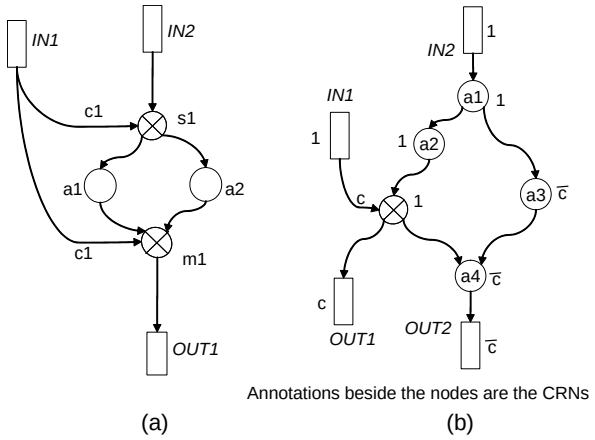

Figure 5. Two simple examples illustrating the notion of CRN

static scheduling so that the state diagram within each software process is derived, as discussed in Section 3.2.

3.4 Condition-aware static scheduling

Scheduling an exDAG begins with annotating the nodes within the exDAG with the conditions under which the nodes should be scheduled. To illustrate this point, we use the example shown in Fig. 5(a). The vertical rectangles in the figure represent *IN* or *OUT* nodes, which are the interfaces the exDAG has with the other processes. In this exDAG, there is a Boolean variable $c1$ controlling both split actor $s1$ and merge actor $m1$. By definition, node $a1$ should only receive a token when $c1$ is true, whereas node $a2$ should only receive a token when $c1$ is false. Therefore, node $a1$ should be annotated with $c1$ as its *conditional readiness* (CRN), whereas node $a2$'s CRN is $\overline{c1}$. A slightly more sophisticated example is shown in Fig. 5(b). In this example, there is also a conditional variable c. All the nodes are annotated with their CRNs. Although node $a3$ is not directly downstream from the split node, its CRN is nevertheless $\overline{c}$. In other words, this node need not be scheduled unless $\overline{c}$ evaluates to true.

The overall *readiness* (RN) of a node is the conjunction of its CRN and its availability readiness (ARN). A node's

ARN is a flag showing the availability of all its incoming data tokens. For computation, split and *OUT* nodes, all the inputs must be present before the ARN is flipped from false to true. For a merge node, only the controlling input and one of its normal inputs are required to be present to have its ARN flipped to true. The *IN* nodes, on the other hand, have their ARN initialized to true since they take inputs from outside the process boundary.

A node is ready to be scheduled if and only if its RN is true. For the example in Fig. 5(b), node $a3$ need not be scheduled at all if c is true, even if its input token is made available after node $a1$ has fired. As one can see, successful and correct computation of the CRN has helped avoid an unnecessary computation. In both the examples in Fig. 5, the CRN of the nodes can be derived by simple reasoning (a systematic procedure is discussed later). For example, in Fig. 5(b), since node $a4$ is directly downstream of the right output of the split node, we can first determine that its CRN is $\overline{c}$. Now, there is no reason to schedule $a3$ if we know that $a4$ would not be scheduled, since $a3$'s only output goes to $a4$. Therefore, $a3$ can also inherit the CRN of $a4$. Given this CRN information, the static scheduler should be intelligent enough to first determine the value of c before scheduling $a3$ and $a4$. That is, knowing that the readiness of $a3$ and $a4$ depends on the value of c, the static scheduler should first determine the value of c before deciding whether to schedule $a3$ and $a4$. Such a static scheduler is therefore *condition-aware* since it is aware of the conditionally-unused operations.

3.5 Energy-aware software architecture synthesis

The software architecture synthesis starts with the initial mapping step. In this step, the initial SAG is generated by performing a one-to-one mapping from the CDFD to the SAG. That is, for every actor in the CDFD, we create a process to only host the actor. Similarly, we create an SAG device component for every device in the CDFD. The same is true for the SAG connectors, which are created by a direct mapping from the edges in the CDFD. The one-to-one mapping makes sure that the initial SAG is valid, as defined in Section 2.

Conversion of the initial SAG into an energy-efficient SAG is done iteratively through a series of software architectural transformations. Essentially, these software architectural transformations seek to reduce the energy consumption by reducing energy wastage due to IPC and context switches. A set of software architectural transformations is illustrated in Fig. 6. In this figure, all the exDAGs shown within the process boundaries do not represent any actual exDAGs. They are there just to illustrate the idea. Also, the graphical notations used in this figure are the same as those in Fig. 4.

Evaluation of the energy gain is done with the help of the OS energy macro-models [12]. An OS energy macro-model is a mathematical function (e.g., an equation) expressing the relationship between the energy consumption of an OS primitive (e.g., a system call) and some predefined parameters. It is derived by fitting a regression model template to the energy consumption data of an OS primitive. The energy data can be collected by low-level energy simulation techniques, or by direct power measurement. Examples of the OS energy macro-models are: (1) OS context-switch energy, $E_{ctx} = 12500$ (nJ) for an *arm-linux* OS, and (2) IPC energy, $E_{pipe\_read} = 690 + 1.65x$ (nJ) for reading x bytes in the pipe_read system call in an *arm-linux* OS.

The energy effects of the software architectural transformations are also illustrated in Fig. 6. These energy effects are expressed in terms of the OS energy macro-models. For example, the energy effect of the *sequential process merging* transformation is given by: $\Delta E = -E_{ipc}(x)N_{ipc} - E_{ctx}N_{ctx}$, where N_{ipc} is the number of times the IPC is invoked, and N_{ctx} is the number of times a context switch occurs.

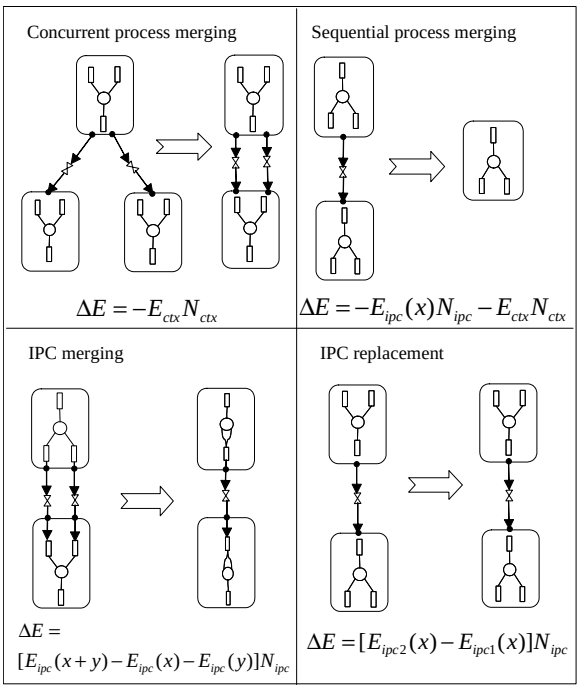

Figure 6. A set of software architectural transformations

4 Experimental Results

As a proof of concept, we built a software synthesis framework based on our proposed methodology. The framework takes as input a CDFD specification and generates prototype C code for fully parallel and energy-efficient software architectures. In order to verify the correctness of our framework and also of the generated C code, we conducted experiments on a number of concurrent behaviors specified in the CDFD format. These CDFDs were designed to mimic the behaviors of some typical embedded systems. They include: `Audio_ctrl2` – audio source selection (CD or tape) and amplification, `Classic` – a synthetic split-and-merge behavior, `Downsamp` – a behavior emulating the down-sampling of a signal, `Stereo` – a behavior that splits a video signal and writes to different output devices, `Synthetic2` – another synthetic behavior, `Webserve` – a behavior designed to mimic the operation of a web-server, `Aware` – a situational awareness system, and `ATR` – a behavior similar to the processing steps involved in an automatic target recognition system.

Processing of the CDFDs and synthesis of the program code are performed on an Intel Pentium III 1266MHz machine running Linux OS. The statistics pertaining to the use of this software synthesis framework are shown in Table 1. In this table, #lines in columns 2, 3 and 4 refer to the number of lines in the CDFD file, the C code of the fully parallel software architecture, and the C code for the energy-optimized software architecture, respectively. Column 5 shows the amount of CPU time taken by the framework to perform the synthesis.

Table 1. Statistics pertaining to the use of the software synthesis framework

Examples	CDFD #lines	Fully Parallel C #lines	Opt. C #lines	Syn. CPU time (s)
Audio_ctrl2	12	576	449	3.07
Classic	8	420	238	2.68
Downsamp	9	451	320	2.55
Stereo	16	750	578	3.03
Synthetic2	16	694	546	2.91
Webserve	7	360	295	2.37
Aware	14	655	531	2.59
ATR	9	452	394	1.48

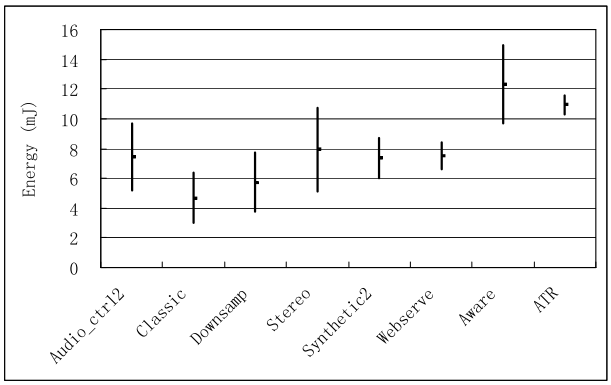

Figure 7. Experimental results showing the energy consumption of the generated C code

We also look at the energy consumption difference between the fully parallel and energy-efficient software architectures. The energy consumption results were obtained by low-level energy simulation of the generated C code. The publicly available simulator we used [13] features an Intel StrongARM target processor and is capable of simulating both the application program and Linux OS (the arm-linux) as a combined image. These energy results are reported in Fig. 7 as a high-low chart. The top of each stick indicates the energy consumption of the fully parallel software architecture, whereas the bottom of the stick indicates the energy consumption of the energy-optimized software architecture. The average energy reduction between these two extremes is 37.9%. It should be noted that there is no performance degradation from the fully parallel software architecture to the energy-efficient one, since our energy-aware synthesis framework merely reduces the energy consumption by reducing energy wastage due to IPCs and context switches. It is also obvious from this chart that the leeway available for software architectural energy optimization is significant.

5 Conclusions

In this paper, we proposed a unified concurrent behavioral representation for high-level modeling of embedded software. The behavioral representation, called the CDFD, models the control/data flow behavior as well as interactions of the embedded software with the hardware devices. We have also proposed a methodology to synthesize a CDFD representation of an embedded software into an actual program code with an energy-efficient software architecture.

To demonstrate the applicability of our software synthesis methodology, we built a software synthesis framework based on the methodology. We also conducted experiments on a number of concurrent behaviors, specified as a CDFD, to verify its correctness. The experimental results also showed that the leeway for software architectural energy optimization is significant. We believe our software synthesis framework will be useful to designers for making a quick energy consumption evaluation of their designed behaviors.

References

[1] V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing Systems*, vol. 13, no. 2, pp. 223–238, Aug. 1996.

[2] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 1997, pp. 72–74.

[3] E.-Y. Chung, L. Benini, and G. De Micheli, "Automatic source code specialization for energy reduction," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2001, pp. 80–83.

[4] W. Wang, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, "Input space adaptive embedded software synthesis," in *Proc. Int. Conf. VLSI Design*, Jan. 2002, pp. 711–718.

[5] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli, "Software synthesis for real-time information processing systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston, MA: Kluwer Academic Publishers, 1994, Ch. 15, pp. 260–296.

[6] F. Balarin and M. Chiodo, "Software synthesis for complex reactive embedded systems," in *Proc. Int. Conf. Computer Design*, Sept. 1999, pp. 634–639.

[7] X. Zhu and B. Lin, "Compositional software synthesis of communicating processes," in *Proc. Int. Conf. Computer Design*, Oct. 1999, pp. 646–651.

[8] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free-choice Petri nets," in *Proc. Design Automation Conf.*, June 1999, pp. 805–810.

[9] J. Cortadella, A. Kondratyev, L. Lavagno, and M. Massot, "Task generation and compile-time scheduling for mixed data-control embedded software," in *Proc. Design Automation Conf.*, June 2000, pp. 489–494.

[10] E. A. Lee, "Consistency in dataflow graphs," *IEEE Trans. Parallel & Distributed Systems*, vol. 2, no. 2, pp. 223–235, Apr. 1991.

[11] H. Gomaa, *Software Design Methods for Real-time Systems*. Boston, MA: Addison-Wesley, 1993.

[12] T. K. Tan, A. Raghunathan, and N. K. Jha, "Embedded operating system energy analysis and macro-modeling," in *Proc. Int. Conf. Computer Design*, Sept. 2002, pp. 515–522.

[13] ——, "A simulation framework for energy consumption analysis of OS-driven embedded applications," *IEEE Trans. Computer-Aided Design*, vol. 22, no. 9, pp. 1284–1294, Sept. 2003.