Trade-off evaluation in embedded system design via co-simulation

Claudio Passerone
Luciano Lavagno
Claudio Sansoè

Dipartimento di Elettronica
Politecnico di Torino
Torino, ITALY 10129
Tel: +39–11–5644111
Fax: +39–11–5644099
email: alcor@polv2k.polito.it
email: lavagno@polito.it
email: sansoe@polito.it

Massimiliano Chiodo

Alta Group of
Cadence Design Systems, Inc.
Sunnyvale, CA 94086
Tel: +1–408–5234632
Fax: +1–408–5234601
email: maxc@altagroup.com

Alberto Sangiovanni-Vincentelli

Department of EECS
University of California
Berkeley, CA 94720
Tel: +1–510–6424882
Fax: +1–510–6435052
email: alberto@eecs.berkeley.edu

Abstract— Current design methodologies for embedded systems often force the designer to evaluate early in the design process architectural choices that will heavily impact the cost and performance of the final product. Examples of these choices are hardware/software partitioning, choice of the micro-controller, and choice of a run-time scheduling method. This paper describes how to help the designer in this task, by providing a flexible co-simulation environment in which these alternatives can be interactively evaluated.

I. Introduction

One of the major problems facing an embedded system designer is the multitude of different design options, that often lead to dramatically different cost/performance results. In this paper we address the problem of trade-off evaluation via simulation, rather than via mathematical analysis. We believe that by providing efficient tools, supporting a variety of target implementation architectures and flexible mechanisms for evaluating design choices, we will help the designer in this difficult task more effectively than by providing relatively inflexible partitioning algorithms.

The problems that we must solve include:

- fast co-simulation of mixed hardware/software implementations, with accurate synchronization between the two components.
- evaluation of different processors and processor architectures, with different speed, memory size and I/O characteristics.
- co-simulation of heterogeneous systems, in which data processing is performed simultaneously to reactive processing, i.e. in which regular streams of data can be interrupted by urgent events. Even though our main focus is on reactive, control-dominated systems, we would like to allow the designer to freely mix the representation, validation and synthesis methods that best fit each particular sub-task.

Since we want to explore trade-offs, we have to use an implementation independent representation. Among the available ones, we use a network of CFSMs, which are Finite State Machines extended to handle integer arithmetic ([CGH+94]). The CFSM model uses a *locally synchronous, globally asynchronous* paradigm, that

- determines exactly how a module behaves whenever it performs a transition, but
- poses little restriction on the speed of communication between modules or on the relative timing of module executions (the "scheduling policy").

By using CFSMs we can leverage the work done in software synthesis so that if we wish to explore a particular hardware/software partition, the software part can be automatically generated optimally. The performance of the system and its correctness can be assessed by running a co-simulation. Because of the semantics of CFSMs and the architecture supported, there is only one resource (microprocessor) that can run code, but there is no limitation on the hardware blocks. Hence all concurrent hardware tasks can be simulated in concurrent mode, while software tasks have to be scheduled onto the processor.

Simulating this implementation requires to know how long each of the software tasks takes, and what scheduling policy we are going to use. To solve the first problem we need an estimation of running times of the tasks.

ASP-DAC '97
0-89791-851-7$5.00 © 1997 IEEE

To solve the second one we need to know the scheduling policy, that can be either given by the user or automatically generated as well. With this information we can use any Discrete Event simulator. It is our intention to leverage maximally what is available and the Ptolemy system ([BHLM90]) is appropriate to solve our problem except for the differentiation among software and hardware tasks, since in the Ptolemy DE simulation each task is considered "hardware". In this paper we show an algorithm that allows to co-simulate CFSM tasks with a given microprocessor architecture and a given scheduling policy without changing any of the Ptolemy code, but by simply having all software tasks implement a distributed scheduler.

Past work in the area of performance prediction and trade-off evaluation has focused mostly on elaborate cost models to guide automated partitioning algorithms, or on co-simulation methods in which a rather detailed model of the processor may be required.

Specifically, co-simulation requires to satisfy two conflicting requirements:

1. accurate modeling of the interaction between software and hardware, which requires in the limit to use cycle-accurate execution of a stream of instructions on a hardware model (emulator) or on a software one (simulator).

2. fast execution of application code in a flexible analysis and debugging environment, which would require in the limit to compile and run the embedded software on a host workstation, somehow simulating its interaction with the hardware and with the environment.

A first class of co-simulation methods, proposed for example by Gupta *et al.* in [GJM92], relies on a single custom simulator for hardware and software. This simulator uses a single event queue, and a high-level, bus-cycle model of the target CPU.

A second class, described by Rowson in [Row94], loosely links a hardware simulator with a software process. Synchronization is achieved by using the standard interprocess communication mechanisms offered by the host Operating System. One of the problems with this approach is that the relative clocks of software and hardware simulation are not synchronized, thus requiring the use of handshaking protocols. This may impose an undue burden on the implementation, e.g. if hardware and software do not need such handshaking since the hardware part in reality runs much faster than in the simulation.

A third class, described in [tHM93], keeps track of time in software and hardware independently, using various mechanisms to synchronize them periodically. If the software is master, then it decides when to send a message, tagged with the current software clock cycle, to the hardware simulator. If the hardware time is already ahead, the simulator may need to back up, which is a capability that few hardware simulators currently have. If the hardware is master, then the hardware simulator calls communication procedures which in turn call user software code.

Our approach is different from those listed above, because:

1. it allows different components of the system to be scheduled independently, without requiring to write a custom environment,

2. software and hardware are simulated together in a unified environment, with the same debugging interface,

3. a bus-cycle model of the target processor is not required, yet a satisfactory level of accuracy is achieved.

The main emphasis of our work is on *speed*, both during simulation (a speed exceeding 1 million clock cycles per second can be achieved on a general-purpose workstation) and when the user changes some architectural parameters (changing the target processor or the hardware/software partition takes about 1 second). This is possible because software is compiled directly on the host workstation microprocessor, and the actual running time on the target microprocessor is determined via estimation, before starting the simulation.

The paper is organized as follows. Section II describes our co-simulation methodology in detail. Section III shows with an example how co-simulation can be used to interactively evaluate the performance of various partitions of a system under various operating conditions. Section IV concludes the paper and outlines opportunities for future research.

II. Co-Simulation methodology

A. The co-simulation environment

Our co-simulation and trade-off evaluation method uses an existing co-design environment for reactive embedded systems, described in [CGH+94], for synthesizing software and hardware, and for analyzing their performance. The *POLIS* system is centered around a single Finite State Machine-like representation, which is well suited to our target class of control-dominated systems. A Co-design Finite State Machine (CFSM), like a classical Finite State Machine, transforms a set of inputs into a set of outputs with only a finite amount of internal state. The difference between the two models is that the synchronous communication model of concurrent FSMs is replaced in the CFSM model by a finite, non-zero, a priori unbounded reaction time. Each element of a network of CFSMs describes a component of the system to be modeled. One of the purposes of co-simulation is exactly to attach timing information to this originally untimed specification, by means of partitioning and profiling.

One of the major strength of *POLIS* is the ability to synthesize both hardware and software components starting from the common model of CFSMs.

- Hardware blocks are mapped into an abstract hardware description format, namely BLIF ([SSL+92]).
- Software blocks are mapped into a software structure that includes a procedure for each CFSM, together with a simple Real-time Operating System. A timing estimator quickly analyzes the program and reports code size and speed characteristics. The algorithm is similar to that used by [PS91], but requires no user input. The estimator is a key component of our co-simulation methodology, because it allows us to obtain accurate estimates of program execution times on any characterized target processor.

Ptolemy ([BHLM90]) is a complete design environment for simulation and synthesis of mixed hardware/software data-dominated *embedded systems*. Here we will concentrate on its simulation aspects. Ptolemy treats the system to be designed as a hierarchical collection of objects, described at different levels of abstraction and using different semantic models to communicate with each other:

- Each abstraction level, with its own semantic model, is called a "domain" (e.g., data flow, logic, ...).
- Atomic objects (called "stars") are the primitives of the domain (e.g., data flow operators, logic gates, ...).
- "Galaxies" are collections of instances of stars or other galaxies. Instantiated galaxies can possibly belong to domains different than the instantiating domain.

Each domain includes a scheduler, which decides in which order stars are executed (both in simulation and in synthesis). In particular, we used the Discrete Event (DE) domain of Ptolemy to implement the event-driven communication mechanism among CFSMs. It has a notion of global time, and the scheduler mantains a global event queue where events are ordered based on their time stamps; at any given instant the event with the smallest time stamp is taken from the queue, and the simulation code of the stars which have that event as input is executed (the stars are "fired"). This domain is event-driven, rather than data-driven as most other domains in Ptolemy, and hence seems the most appropriate for our purposes.

B. CFSM co-simulation in Ptolemy

The Ptolemy scheduler in the DE domain fires stars as if they were executed concurrently. Thus it does not directly provide a way to simulate CFSMs implemented in software and running on a limited amount of computational resources (in this paper we assume that a *single CPU* is available). Our goal was to modify the DE scheduler behavior without changing its code, to maintain compatibility with the original version. Thus we let the scheduler fire stars in its own preferred order, but every star may or may not actually execute the main part of its code, based on *global* information which characterizes the shared resources. All this is accomplished in a transparent way so that the Ptolemy scheduler sees a world of concurrent stars, while the software stars see the *POLIS* scheduling policy.

Having met this goal, it is now possible to simulate in Ptolemy a system designed using *POLIS*: this is accomplished by generating a proper description of every CFSM and loading it into Ptolemy together with the network of interconnections. It is then possible, through the nice graphical interface provided by Ptolemy, to evaluate trade-offs between hardware and software solutions, and to visualize the overall system behavior.

The detailed design flow for the co-simulation and trade-off analysis phases is as follows:

1. The control/data flow graph of every CFSM in the system specification is built, and the corresponding C code is generated (as described in [CGH+95]). The C code also includes run time estimations for each C code statement, based on information derived from benchmark analysis of the target processor ([SSV96]).
2. The Ptolemy language source code for every CFSM is generated. This in turn includes the C code generated by the previous step.
3. All CFSMs are loaded into Ptolemy. Each of them is a single star, with input and output portholes corresponding to CFSM inputs and outputs.
4. The network of interconnections between stars (CFSMs) is created in the Ptolemy environment.
5. Each star is assigned (by interactively modifying one of its properties or one of the properties of a galaxy above it in the hierarchy):
   - an implementation, either software of hardware, and
   - a priority, used by the software scheduler.

   Hardware stars run concurrently and terminate in a single clock cycle. Software stars are mutually exclusive, and use the run time estimation to determine how long it takes to emit outputs and complete firing of a single transition of the CFSM. Since the underlying model for both hardware and software is the same, changing the implementation can be done at run-time by just accumulating or not the timing information.

6. A single system-wide parameter (also modifiable interactively) describes which one of the pre-characterized processors must be used for cycle counting. This provides a mechanism to easily change the target processor for a given set of simulation stimuli, without the need to re-analyze or re-compile the specification. In the same way it is possible to specify the scheduling policy best suited for the given application. If the scheduler is priority-based, then it can use the priority level assigned to each star.

7. The simulation is started with appropriate stimuli generators and output monitors, to check the behavior of the system. Multiple simulations can be compared to evaluate timing constraint satisfaction, run time, processor occupation, and other interesting pieces of information.

   For example, in Section III we will describe how the CPU utilization can be monitored, to detect and analyze potential overload conditions.

C. Scheduling policy implementation

The solution that we have chosen *does not require to modify the DE scheduler*. From now on, we will call *software scheduler* our own scheduling policy, implemented by an automatically generated procedure on top of the Ptolemy DE scheduler. Each simulation cycle (identified by an integer number, and directly corresponding to a simulated system clock cycle) is divided into three phases:

1. **Request phase**, in which all stars receive events, and request from the software scheduler access to the processor, re-scheduling themselves at the grant phase.

2. **Grant phase**, in which only one star is granted access to the processor and executes its user code, while all other enabled stars re-schedule themselves at the update phase. The identifier of the star which currently runs on the processor is kept in a shared variable called *star_ack*. This star also computes the time at which the processor will become available again, in a shared variable called *next*, by accumulating estimated clock cycles during the execution of user code and RT-OS calls.

3. **Update phase**, in which all enabled stars re-schedule themselves at the next time the processor is available.

The request phase is characterized by an integer simulation time, while the grant and update phases are characterized by fractional simulation times. All events are received and emitted at integer times.

Scheduling is thus performed in the following way. Whenever a software star receives an event:

- If the current time is greater than or equal to *next*, then the processor is available:
  - If the current time identifies a request phase, then the star sends the request to the software scheduler, and re-schedules itself at the next grant phase.
  - If the current time identifies a grant phase, then the star must check the variable *star_ack* to know if it has been chosen by the software scheduler. In this case, it executes the user-specified part of its own code, accumulating estimated clock cycles depending on the sequence of instructions that is executed, and sets the variable *next* according to the clock cycle estimate. Otherwise it must re-schedule itself again at the next update phase (to make sure that the selected star has had time to execute its code and update the variable *next*).
  - If the current time identifies a grant phase, then the star reads the variable *next* and re-schedule itself to try to get the processor at that time.
- If the processor is not available, but the priority of the star is greater than that of the currently executing star, and *the chosen scheduling policy allows interrupts* (i.e. it is pre-emptive), then an interrupt occurs. The variable *next* is incremented by the estimated execution time of the interrupting star, and the interrupted one is prevented from emitting output events until the end of the interrupt. Interrupts may be arbitrarily nested, and can only cause delays in the interrupted stars, without changing their behaviour (input variables to stars are buffered in our software implementation scheme, to improve the predictability of the system behavior).
- If the processor is not available and the priority of the star is less than or equal to that of the currently executing star, or if the software scheduling policy is non-pre-emptive, then the star must re-schedule itself at time *next*. Hence, stars with the same priority level do not interrupt each other.

The communication queues among stars are forced to hold at most one event, to match the CFSM communication model using one-place buffers. This means that events may be overwritten, if they are emitted twice without being detected. This is *legal* in the CFSM model of computation, but can optionally be *logged to a file*, because often losing events means violating timing constraints and is interesting for the designer.

The implementation (hardware or software) of each star can be dynamically and independently changed during a Ptolemy session, by just updating a parameter of the star or of a galaxy enclosing it in the hierarchy. Thus it is possible to experiment with different solutions in a very

straightforward manner and in a short period of time. In fact, it is not necessary to rebuild the system simulation model, but it is sufficient to execute a new simulation of the same model.

III. An application example

We consider an application from the automotive domain: a dashboard controller. It takes as inputs pulses coming from the sensors on the wheels and on the engine, and data about fuel level and water temperature. It processes the inputs and drives, using pulse-width modulated signals, a set of dials.

We evaluated different implementation choices, under various possible operating conditions. The timing constraints, which in this case are relatively soft, derive mainly from the need not to miss any incoming pulse. For the engine this means up to 5000 pulses per second, while for the wheel this means up to 16 pulses per second. Outputs must be produced at a frequency of at least 100 Hz and with a maximum jitter of 100 microseconds to drive the gauge coils. In this case, we first assumed to use a Motorola 68HC11, because the timing estimate available from synthesis (379 clock cycles at most for the most time critical task, the engine pulse recorder) showed that we could hope to satisfy the requirements with a 1 MHz processor.

The system was modeled using 13 CFSMs, each specified in ESTEREL, for a total of 2500 lines of source code and 10 KBytes of compiled object code for the final implementation. Their interconnection was described graphically with the Ptolemy user interface, as illustrated in Fig. 1. One CFSM is devoted to input conversion from level to pulse, three CFSMs derive timing events from the base time reference, four CFSMs filter and normalize the data, one CFSM converts potentiometer readings to fuel level (taking into account the shape of the tank), and the remaining four CFSMs perform the PWM conversion. Scheduling was performed by hand, due to the simplicity of the system, resulting in an assignment of two priority levels to CFSMs. The highest level was assigned to the CFSMs that had to handle input events, and corresponds, in practical terms, to interrupt-driven I/O without interrupt nesting.

We first tried to implement everything in software, then moved part of the components to a hardware implementation in order to satisfy the timing constraints. The PWM converters were an obvious choice for hardware implementation, because their low jitter constraints made a software implementation on a 68HC11 infeasible. Fig. 2 shows the priority level of the star currently executed on the processor as a function of time[1]. The processor utilization is still fairly high, especially considering that the simulated car speed was about 50 Km/h. Hence we moved also the timing event generators to hardware (a typical choice in embedded systems, in which timing functions are often performed by special-purpose timers which are part of standard micro-controllers). Fig. 3 shows the priority level in this second case. The user interface uses the standard constant specification mechanism provided by Ptolemy, and allows the designer to change all the architectural parameters without re-compilation. Currently supported parameters are:

- CPU type, clock speed, scheduler type for the whole system,
- implementation (hardware or software) and priority (used only for software stars) for each star or hierarchical star group (galaxy).

The performance of the simulator is very high, especially if there is no component which is active at every clock cycle, because in that case we can exploit the inactivity of the system. The dashboard example is ideal in this respect, because the highest frequency input events occur about once every 100 clock cycles (assuming a 1 MHz clock). The results for the dashboard simulation, using various types of architectural choices, are reported in Table I. In this case, we used estimated execution times for a Motorola 68HC11 micro-controller with 1 MHz clock speed, and a MIPS R3000, with 1 MHz and 10 MHz clock speed. The partitions shown in the column labeled "Part." are respectively:

- SW: all modules are in software,
- HW/SW: the PWM drivers and timing generators are in hardware, and the rest is in software,
- HW: all modules are in hardware.

The column labeled "Graph." shows when the graphical priority display (useful for debugging the software scheduler) is used. All CPU and User times are in seconds, and were obtained on a SPARCSTATION 10 with 16 Mbytes of RAM, simulating 20 million clock cycles (except for the 10 MHz MIPS, for which 200 million cycles were simulated). The user time required to restart a simulation when the partition or the target processor are changed is about 1-2 seconds.

The execution time for the all-software partition on the 68HC11 is very high because the processor does not meet the timing constraints, and hence the simulation is interrupted very often to log the information about missed deadlines on a file.

The simulation performance (without graphics output, that significantly slows down the system) is around 1 million clock cycles per second. This speed, which can be achieved thanks to the extremely low overhead imposed by our cycle counting technique, is sufficient in many cases to run simulations almost at the same speed as the real target system (*virtual prototyping*).

[1]A value of -1 means that the processor is idle, 1 is the highest level, each vertical bar represents a context switch.

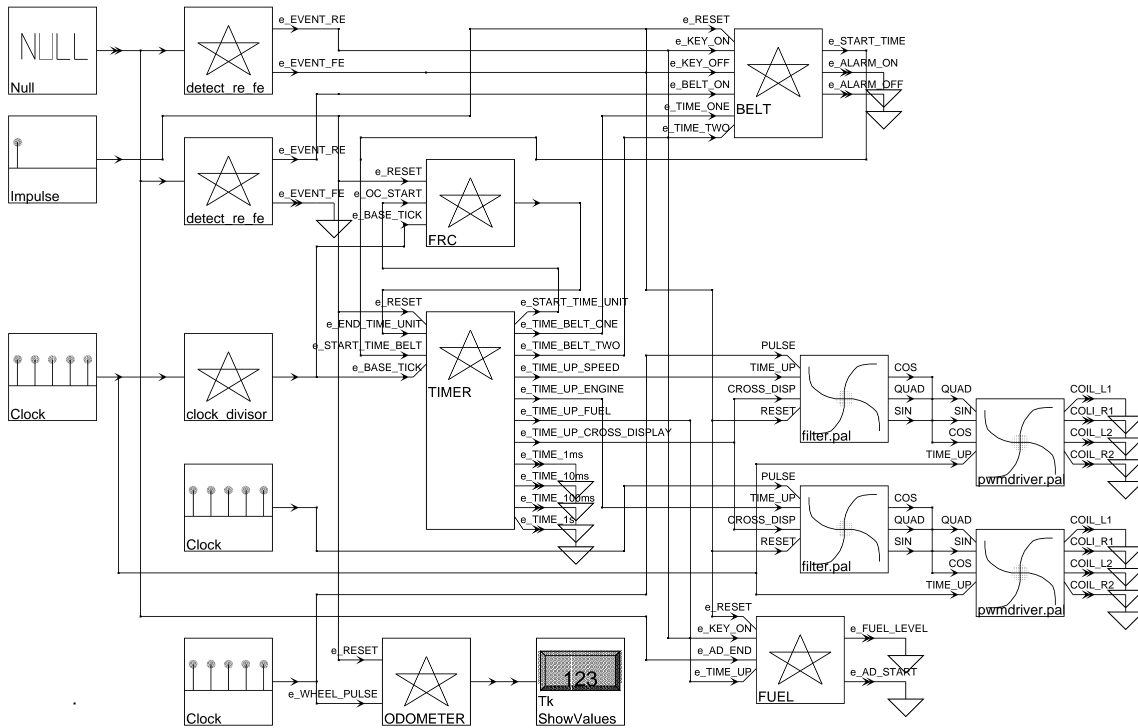

Fig. 1. The dashboard controller netlist

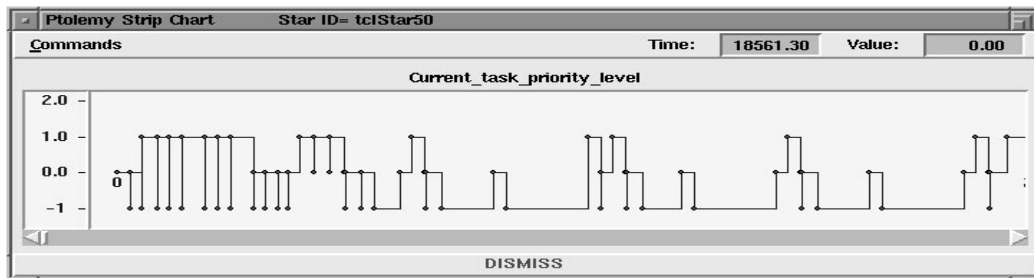

Fig. 2. Processor utilization analysis with few hardware components

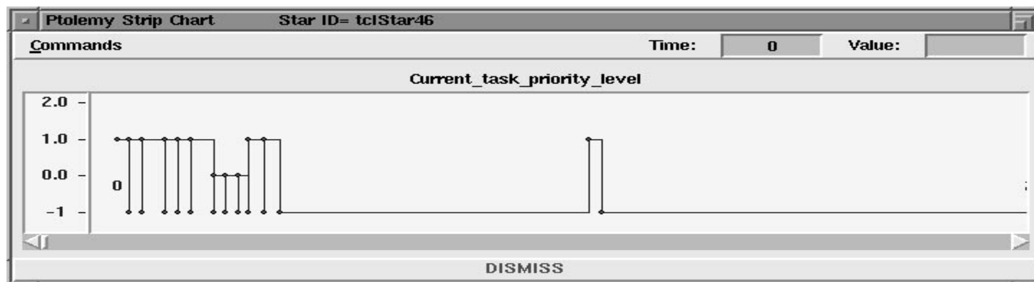

Fig. 3. Processor utilization analysis with more hardware components

TABLE I
SIMULATION SPEED FOR VARIOUS TYPES OF SYSTEM PARTITIONS

Target Proc.	MHz	Part.	Graph.	Time	
				CPU	User
HC11	1	SW	No	> 1000	> 1000
HC11	1	HW/SW	No	30	38
HC11	1	HW	No	25	27
MIPS	1	SW	No	161	162
MIPS	1	HW/SW	No	23	25
MIPS	1	HW	No	25	26
MIPS	10	HW/SW	No	23	23
HC11	1	HW/SW	Yes	202	205
MIPS	1	HW/SW	Yes	258	270

IV. Conclusions and future work

In this paper we have shown that fast co-simulation can be done at the early stages of a design, for partition evaluation and functional verification purposes. The methodology relies on the use of constrained software synthesis, that permits easy run time estimation for a target processor, and of a powerful co-simulation environment built in the Ptolemy system.

We noticed, by profiling the simulator code, that over 90% of the time (when the graphic output is not used) is spent executing the DE scheduler code. This means that a faster simulator could be obtained by re-writing the Ptolemy DE Scheduler to take into account the required behavior of CFSM stars, and eliminate the overhead introduced by the re-firing method. On the other hand, this option may not be desirable for reasons of compatibility, both with future versions of Ptolemy, and with other simulation domains within Ptolemy.

In the future, we would like to allow the designer to create more than one software partition, thus simulating multiprocessor environments, and to specify hand-estimated execution times for software modules that were not synthesized using POLIS (e.g., data-intensive modules designed using Ptolemy).

References

[BHLM90] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[CGH+94] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Hardware/software codesign of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.

[CGH+95] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of software programs from CFSM specifications. In *Proceedings of the Design Automation Conference*, June 1995.

[GJM92] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the Design Automation Conference*, June 1992.

[PS91] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.

[Row94] J. Rowson. Hardware/software co-simulation. In *Proceedings of the Design Automation Conference*, pages 439–440, 1994.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[SSV96] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation models for hardware/software codesign. In *Proceedings of the Design Automation Conference*, June 1996.

[tHM93] K. ten Hagen and H. Meyr. Timed and untimed hardware/software cosimulation: application and efficient implementation. In *Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.