International Journal of Electronics Engineering, 1(2), 2009, pp. 203-207

Hardware and Software Co-Synthesis Environment for Embedded Systems

Rashmi Priyadarshini & B. S. Chawla

Department of Electronics and Communication Engg.,
Indira Gandhi Institute of Technology, Guru Gobind Singh Indraprastha University, Delhi

Abstract: *This paper presents a hardware and software co-synthesis environment for embedded systems. The environment uses two graphical formalizations as specification languages and synthesizes code for a multiprocessor rapid prototyping board. The two major problems discussed in this paper are: (a) realization of an efficient distributed execution of the specific system (b) development of an automated interface code generation for hardware and software of the system under design.*

1. INTRODUCTION

Embedded systems are applied in every area of life and their importance increases rapidly. But the development of such systems is often costly and time intensive, especially when their functionality can only be tested in the late phases of the development process. For example, more than 50% of all complaints about electronic devices in automobiles are due to conceptional error made during the design and specification phases. *Rapid Prototyping board* can help to detect and correct defective specifications thereby preventing expensive design-iterations during the development of a system.

The paper is mainly concerned with those embedded control applications (ECAs) that require fast reaction to asynchronous external events. Examples for such applications are collision avoidance systems or controllers for fast run manufacturing cells.

One part of this system is a "universal prototype" hardware architecture that consists of an FPGA field and several processor nodes. Each node is equipped with a multi-threaded implementation of the SPARC processor called MSPARC, which is optimized for embedded control applications. However, this paper concentrates on the second part of the system, a hardware and software co synthesis environment where two graphical formalism, *real time symbolic timing diagrams and statecharts,* are used to specify ECAs and combine them with tools for generating interfaces between hardware and software components.

2. ARCHITECTURE OVERVIEW

The architecture currently developed in the project is shown in Figure 1. Each node contains a MSPARC (Multithread Scalable Processor Architecture)-processor and a 2nd-level cache. SPARC (Scalable Processor Architecture) is a RISC microprocessor instruction set architecture originally designed in 1985 by Sun Microsystems. The dual ported RAM (DPR) is used to communicate with external devices and to schedule context-switching. Unlike other architectures, the FPGAs (Field Programmable Gate Array) are viewed as the system's master and the processors as coprocessors to the FPGAs.

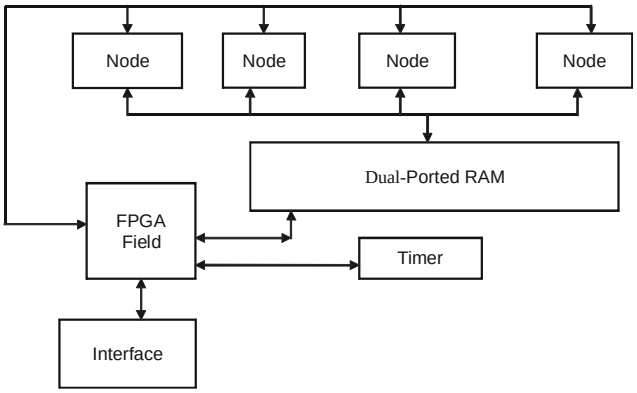

Figure 1: The Hardware Architecture

The MSPARC is compatible to a standard SPARC processor [14], and additionally supports multi-threading for up to four contexts on chip. The MSPARC has a five stage pipeline and an 8KB on chip instruction cache. The cache is statically divided into four parts, one for each context. Context switching is executed by hardware and can be achieved in one processor cycle. Because the new context starts with an (almost) empty pipeline, the worst case penalty for context switching is five cycles. This fast switch mechanism is attained by duplicating the important register file. A round robin strategy is used to determine the context that will be switched to.

The MSPARC also provides an embedded control mode, in which the environment (in our case the FPGAs) can control with active signals (one for each context) which

*Corresponding Author: dhirajsangwan@gmail.com

context to switch to, and a single switch signal is used for each processor to initiated context switching. Additionally, threads can signal the termination of computation to the environment with done signals. If there are no contexts marked as active by the FPGAs, the MSPARC will halt execution. [1]

The actual realization of this architecture is divided into four stages. Stage 1 consists of an FPGA-board connected via a serial interface with a workstation [2]. Stage 2 consists of connection board with a dual–ported RAM and some registers for thread control. The interface of this board to the FPGAs will be exactly the same as it would be if four MSPARC–processors were connected to them. In stage 3, one MSPARC–processor is connected to FPGA board, whereas in stage 4, the complete multiprocessor architecture will be built.

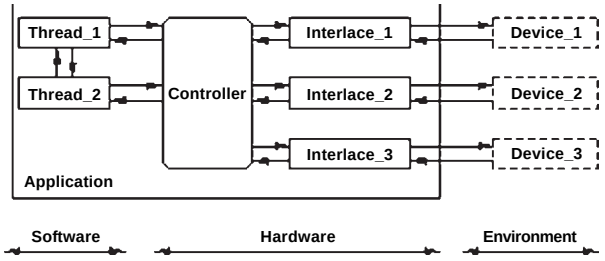

Figure 2: Basic Template for ECAs

3. SPECIFYING REACTIVE SYSTEMS

When hardware/software applications with time – critical computations are considered, a close mapping between the design architecture of the system under construction and the actual hardware architecture is necessary in order to achieve the required performance. This section presents some basic design rules and concepts found appropriate for our environment.

The activity of the application is split up into several sub activities–the *components* of the system-which are be classified with regard to their responsibilities into following types*: interface components* are responsible for connecting the systems with external devices, sensors or actuators. Interface components are always implemented in hardware, since in this architecture only the FPGA communicate with the environment. *Thread component* are activities responsible for performing the computation parts of the application. The *control components* supervise all activities in the system.

3.1. Events

Whenever a situation occurs in an embedded control system that requires an action from the application, this situation is called an *event*. Frequent events are sensors sending new values, timers reaching zero or variables being written. Typical actions associated to events are polling of data from sensors, computing a new system state or computing new actuator values.

The three kinds of events are considered: the first type does not require elaborate computation but merely causes updates of actuators of the systems state (control dominated reactions to events) and /or events for which software would be to slow to perform these updates. These events are usually handled by interface components. The second kind of events requires complex computation and is therefore handled by threads. The start-up time of the appropriate thread is part of the response time to this event and is not negligible on usual architectures due to expensive software scheduling and context switching .These threads will be assigned to their own context on the MSPARC-processors in the architecture resulting in startup times of just one cycle [1]. The third kind of events requires handling by software, too, but the time constraints are such that the startup time of threads can be tolerated. For scheduling and dispatching purposes this context will be used in the same way as the (only) context in a single-threaded uniprocessor system would be. [4]

3.2. Specification

Statecharts are used to *d*esign and implement threads and interface components [8]. State chart is a graphical specification formalism, which is widely used to design reactive systems. The control component can be specified either by statecharts or by *real time symbolic timing diagrams* (RTSTD) [5]. RTSTD is graphical specification formalism similar to the well known diagrams but in contrast to other formalized timing diagram approaches. [2]

RTSTD consists of a set of waveforms and constraints. A waveform defines a sequence of expressions for a signal, which describes a possible temporal evolution. The point of change from validity of another expression is called a symbolic event. Events are normally not ordered across different waveforms, but constraint arcs can be used to impose such an ordering. A constraint can be a *simultaneity constraint* (two events must occur simultaneously), a *conflict constraint* (two event may not occur within a given time-interval), a *leads to constraint* (an event e2 may not occur later than time *t* units after an event e1), or a *precedence constraints* (event e2 may only occur if preceded by event e1 within a given time interval). *Strong constraints* express requirements which must be satisfied by the component under design, while *weak constraints* express an expectation on the behavior of the environment. Figure 3, shows a simple handshaking protocol specification, where the strong constraints are printed as black arcs while weak constraints are printed in grey.

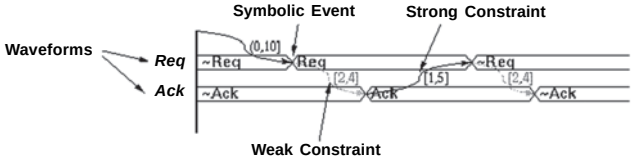

Figure 3: A Real-time Symbolic Timing Diagram

RTSTDs are more expensive and concise when dealing with complex timing constraints than statecharts. RTSTDs is very versatile, rendering them a natural choice for specification of control components.

3.3. Controlling Thread Execution

To specify control of threads with timing diagrams, special type thread is added to the language of RTSTDs, and some predicates are defined start, stop, started, stopped for variables of type thread (Figure 3). start (*t*) does not imply started (*t*), and neither does stop (*t*) imply stopped (*t*). If *t* is stopped, then start (*t*) causes *t* to change its state to started eventually, and analogously causes stop (*t*) the thread *t* to stop, but not necessarily in the next step.

3.4. Scheduling

In general, the specification of thread control with RTSTDs induces only a partial order between threads. The task of the scheduler is to refine this order for the *n*-processor MSPARC architecture.

The simplest approach for this refinement is to execute threads in the order of their activation. This scheduling strategy, called naive scheduling, has the advantages that the user does not have to worry about scheduling at all, since the scheduler is synthesized without user intervention as part of the timing diagram synthesis. A disadvantage of this approach is that certain specifications become unimplementable simply because the naïve scheduling could choose an order that would lead to a violation of timing constraints.

Another approach, called extended built-in scheduling, is to let the built-in scheduling mechanism of the MSPARC do the scheduling between all threads allocated on one node.Internode scheduling is done as before. Advantages and disadvantages of the approach are more or less the same as in the naïve approach.

The third scheduling strategy is called speed scheduling .In addition to the activation of threads a minimal (required) execution speed for them is specified by the user with a special predicate speed (*t*, *s*) which assigns speed s to thread *t*. Specifications which are not implement able under the naïve scheduling strategy become implement able if the sum of execution speeds of parallel threads allocated on one processor dose not exceed the CPU's total execution speed.

Currently both naive and speed scheduling are supported the formal allowing for a comfortable specification of less stringent real time specification the later being better suited for very time–critical and highly concurrent specifications.

4. CODE GENERATION

Figure 4, shows the code generation process. The design starts with the statechart and timing diagram specifications. A closer look at synthesis steps are as follows:

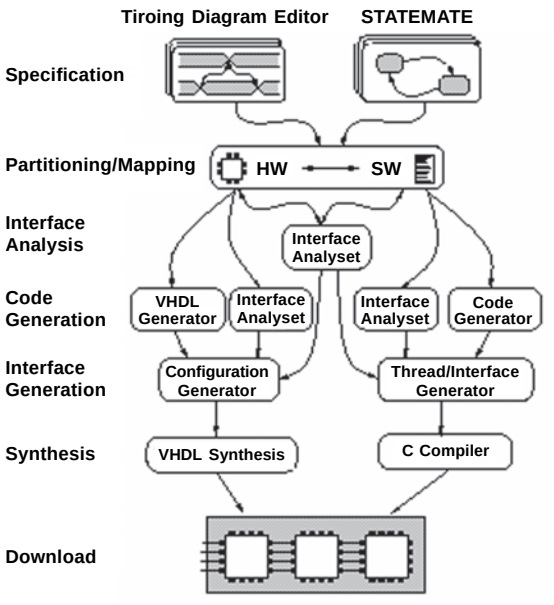

Figure 4: Code Generation Process

4.1. Partitioning

In the partitioning step the user annotates each activity of the activity-chart with a thread. These steps are not automated. The manual partitioning at this design level gives the best result.

After threads and hardware components are identified, they must be mapped to the architecture. As first step, mapping of hardware components to FPGAs and software components to contexts will be done manually.

4.2. Interface Analysis

The interface analysis step is used twice in the code generation process. In the first stage, the dataflow between components is determined. The result of this stage is used to guide the mapping tools to keep communication overhead low.

In the second stage, back-annotation from the mapping tools to produce detailed symbol tables as input for the interface generators is used. These symbols tables contain information about the dataflow between the components, about the mapping of components to the architecture and about resource allocation on the dual-ported RAM.

4.3. Code Generation

In the semantics of statecharts, all parallel automata are executed step-synchronous. This model is obviously not suitable for distributed systems. Since performance is critical for the given target applications, communication and synchronization overhead as far as possible should be reduced. Therefore, an asynchronous execution model for state chart/timing-diagram specifications is used. For the threads, the State mate code generator to synthesize code to wrap this code and produce hardware-controllable threads is used.

To synthesize thread control expression from RTSTDs, the ICOS synthesis tools create for each thread *t* an output signal active_t and an input signal done_t. During the mapping and the analysis stage ,a symbol table is produced where these signals are declared as thread control signals and mapping to specific pins of the selected FPGA. The interface generator then uses this symbol table to map these signals to the ports of the MSPARC processor on which *t* is allocated, and produces additional code to set the switch signal of this processor. If *t* should be controlled by a scheduler, than active_t and done_t together with additional scheduling attributes have to be connected with the hardware part of scheduler**.**

4.4. Interface Generation

The hardware and software interface establishes the dataflow from threads to the hardware components and vice-versa. In our architecture, the DPR is used for hardware and software communication .The interface analyzer computes the set of communication variables and allocates memory for these variables in the DPR.

Communication is always initiated by a value change of a communication variable. If a value change occurs, a callback function is invoked to update the DPR with the new value of the variable. For the software components, invocation of callback functions is managed by the State mate C-code machine, where as in hardware, observer processes are used to detect change of values.

Threads update their local memory at the beginning of their execution by copying the relevant data from the DPR. The controller updates its local memory when threads terminate. To accomplish this, the interface generator has to create update-code for each thread.

5. A Case Study

Our first case study is a simple air conditioning system for automobiles, completely specified with activity chart and statecharts. Figure 5, shows the top-level activity-chart.

The system controls four stepper motors which are connected with a serial bus, called MI-bus. The **MI_Controller** component is responsible for sending and receiving messages, **SequenceGen** creates message sequences, which set the motors into a given position. These two activities are the interface components of the system. **ProgControl** is connected with the MI interface and several sensors and controls five air-condition programs, the threads of the system. The tasks of these programe are to calculate a new position for the steppermotors. They get their inputs from the sensors and the program controller.

In this case study we used the FPGA board which was connected with a workstation, and a serial interface to realize data-communication between workstation and FPGA board. Since this interface does not allow any FPGA-initiated actions, dataflow from hardware to software had to implement by polling and additional activity polls the contents of the interface every *n* steps and updates the memory of the State mate machine accordingly. Since we do not use multi-threading at this stage, we implemented the program controller as a software component.

6. CONCLUSION

In this paper, co synthesis approach for rapid prototyping of embedded system is presented The main features of this system are, first, that programming is done completely with graphical specification languages, second, that interface code between hardware and software components is synthesized automatically from the specification, and third, that a multi-processor multi-threaded hardware architecture for short response times is used.

REFERENCES

[1] T. Bienmiiller, A. Metzner, and A. Miksehi, ‘Traps and Fast Context Switches in a Multithreaded Environment”. A SPARC, Technical Report, University of Oldenburg, Germany, (1997).

[2] G. Borriello, ‘Formalised Timing Diagrams’, In the Europian Conference on Design Automation. IEEE Computer Society Press, (1992).

[3] R. Ernst, J. Henkel and T. Benne, ‘Hardware/Software Co-synthesis for Microcontrollers’, *IEEE Design and test Computers*, (1993).

[4] K. Feyerabend and B. Josko, ‘A Visual Formalism for Real Time Requirement Specification’, (1997).

[5] K. Feyerabend and R. Schlor, ‘Hardware Synthesis from Requirement Specification of Computn.’ (1996).

[6] M. Franzle, K. Liith, ‘Compiling Graphical Real-Time Specification into Silicon’, (1998).

[7] R. Gupta, G. De Mithil, ‘Hardware/Software Co-synthesis for Digital System’ (1993).

[8] D. Harel H. Lachover, A. Naamad, A. Pnneli, M. Polite R. Sherman, M. Trakhten, ‘STATEMATE: An Environment for the Development of Complex Reactive Systems’, (1990).

[9] D. Harel, ‘Statecharts: A Visual Formalism for Complex Systems’. (1987).

[10] G. Koch, U. Kebschiill, W. Rosenstiel, ‘A Prototyping Environment for Hardware/Software COBRA project’. (1994).

[11] F. Korf and R. Schlor, ‘Interface Controller Synthesis from Requirement Specifications’. (1994).

[12] K. Liith, A. Metzner, T. Peikenkamp, and J. Risan, ‘Events Approach to Rapid Prototyping for Embedded Control Systems’. (1997).

[13] A. Mikschi and W. Damm, ‘MASPARC: A Multithreaded Spare’. In L. Bonge, P. Fraigniand, A. Mignotte and Y. Robert, Parallel Processing, (1996).

[14] Sparc International Inc. The Sparc Architecture Manual Version 7, (1990).

[15] L. Stoica, H. Abdel Wahab, and K. Jeffay, 'On the Duality between Resources Reservation and Proportional Share Allocation, (1997).

[16] Karsen Luth Jurgen Niehaus, Thomas Peikenkamp Carl von Ossietzky University, Oldenburg. HW/SW Co-synthesis using State Charts and Symbolic Timing Diagrams, (1990).

[17] S. Bhattacharya, 'HW/SW Co-synthesis using DSP Microprocessor', (2001).

[18] R. Niemann, 'Peter Marwedel'. HW/SW Partitioning using Integer Programming, (1996).

[19] Sparc International Inc. 'The Sparc Architecture Manual Version 8'. Prentice Hall Englewood Cliff, NJ, (1992).