Automatic Generation of a Real-Time Operating System for Embedded Systems

Extended Abstract

Felice Balarin* Massimiliano Chiodo† Attila Jurecska‡ Luciano Lavagno*
Bassam Tabbara§ Alberto Sangiovanni-Vincentelli§

December 18, 1996

1 Introduction

Embedded systems are typically implemented as a set of communicating components some of which are implemented in hardware and some of which are implemented in software. Usually many software components share a processor. A *real-time operating system* (RTOS) is used to enable sharing and provide a communication mechanism between components. Commercial RTOSs are available for many popular micro-controllers. Using them provides significant reduction in design time and often leads to better structured and more maintainable systems. However, since they have to be quite general, they are not efficient enough for many applications, either in memory usage or in run times. Thus, it is often the case that RTOSs are hand coded by an expert for a particular application. This approach is obviously slow, expensive and error-prone.

In this paper we propose an alternative where a RTOS is automatically generated based on a high-level description of the system. RTOSs created in our approach offer an ease of use comparable to commercial RTOSs, and yet since they are generated for a specific example, they can be optimized based on the same information used to optimize hand-written code.

We have implemented our approach within POLIS [2], a system for HW/SW co-design of embedded system. To evaluate the POLIS generated RTOS we have developed a prototyping environment which we use to compare POLIS against a commercial operating system.

The rest of the paper is organized as follows. In section 2 we review the formalism for high-level specification of systems used in POLIS, and then we describe RTOS generated by POLIS in section 3. A description of prototyping environment and initial experimental results are presented in section 4, while final comments are given in section 5.

*Cadence Berkeley Labs, Berkeley, CA USA
†Alta Group of Cadence Design Systems, Sunnyvale, CA, USA
‡Magneti Marelli, Torino, Italy
§Dept. of EECS, University of California, Berkeley, CA, USA

2 Codesign Finite State Machines

In POLIS, systems are represented as networks of objects called *Codesign Finite State Machines* (CFSMs). Each CFSM *emits events* in response to the events occurring in the environment, or to the events generated by other CFSMs in the system. Which events should a CFSM emit in response to certain input events is determined by a (finite) set of rules called a *transition relation*.

Events are the basic observable entities that define the behavior of the system. More precisely, the behavior of the system is the set of sequences of events that CFSMs in the system can produce. Events can have a *value*, e.g. the event with name "keyboard" could occur every time the user hits a key with a value belonging to the ASCII set. The value of an event is persistent, e.g. the value of the "keyboard" event is defined at *all times* to be the last key hit.

A CFSM is like a "classical" FSM, because both transform a set of inputs into a set of outputs by using only a finite amount of internal state. The difference from the "classical" FSM (that will be called just FSM in the following) is that our model has no implied "synchronous" hypothesis. The standard definition of interaction between FSMs assumes that all the FSMs change state exactly at the same time. This can be very different from the actual behavior of a mixed hardware/software system, in which software components can take hundreds of clock cycles. In contrast, CFSMs operate concurrently and independently by repeating the following four phases:

1. idle,

2. detect input events,

3. transition, according to which events are present and match a transition relation element,

4. emit output events.

Phases 1, 2 and 4 can have a duration between zero and infinity, while phase 3 takes *at least* some time. In either case, the duration of phases is independent of any other CFSM in the system.

In phase 2 a CFSM must be able to detect all the events that have occurred since its last transition. However, if some event occurs more than once in that period of time, only the most recent occurrence can be detected. A possible implementation of these rules are queues of length 1. Events are emitted by writing to a queue, and detected by reading from a queue. Writing to a full queue overwrites the existing content. Each CFSM has its private input queues. Thus, it is quite possible that two CFSMs detect the same event at different times (of course, both of those time must be after the actual occurrence of the event).

In POLIS, CFSMs are also the basic unit of implementation. Each CFSM is implemented as a module in one of the following three ways:

1. Some CFSMs are implemented in synchronous hardware. Hardware implementations of CFSMs are denoted HW CFSM.

2. Some CFSMs are implemented in software. For every such CFSM POLIS generates a C function (denoted SW CFSM) with calls to detect and emit functions (that are defined by the RTOS).

3. Some CFSMs are implemented by micro-controller peripherals. Most of the modern micro-controllers come with powerful built-in peripherals. Programmable timer units are probably the most common example. Implementations of CFSMs with these units are called MP CFSMs. For them, POLIS generates calls to library functions that program the units properly.

3 Real-time Operating System synthesis

In this section we describe the RTOS generated by POLIS for a given network of CFSMs. Its major responsibilities are to:

1. provide communication mechanism (i.e. define detect and emit functions) between SW CFSMs and other SW CFSMs, HW CFSMs and MP CFSMs,

2. coordinate the execution of SW CFSMs.

3.1 Implementation of events

For each SW CFSM and each event it is sensitive to, the RTOS maintains a flag which is set when the event occurs and reset when that SW CFSM makes a transition. Emitting an event thus requires setting these flags for all potential consumers. Detecting an event is implemented as a macro that checks if a flag is set.

The value of an event is communicated through a shared variable. While RTOS distributes information about event occurrence to all the detecting CFSMs, it keeps only a single copy of the event value (for obvious, memory-saving reasons).

The event emission and detection capability described above is sufficient for communication between SW CFSMs. For communication with other types of CFSMs it is extended as follows:

SW CFSM $\rightarrow$ HW CFSM If an event emitted by a SW CFSM can be detected by a HW CFSM, then it is assigned an output port and the emit function is extended to generate a pulse on that port (by writing to it 1 and then 0). The output port is assigned from a pool that includes actual I/O ports of the micro-controller, as well as additional memory-mapped ports (for which POLIS also generates appropriate hardware).

SW CFSM $\rightarrow$ MP CFSM In this case the emit routine includes a call to a library function that alerts a peripheral that a particular event has occurred (e.g. that a timer has been requested to start).

HW CFSM $\rightarrow$ SW CFSM There are two ways that an event emitted by HW CFSM can be communicated to a SW CFSM: by polling or by interrupts, as specified by the user. For each interrupt-driven event, RTOS assigns an *abstract interrupt vector*. Micro-controller-specific information is then used to map abstract interrupt vectors to actual interrupt vectors of a target micro-controller. This task is trivial if there are fewer abstract than actual vectors, but if that is not the case, some multiplexing hardware and software has to be generated. In this way, we can deal with interrupts in a fairly machine-independent fashion. For every assigned abstract interrupt vector an *interrupt service routine* (ISR) is generated. By default, this routine contains only a call to appropriate emit function. However, a user may require that the ISR also includes calls to SW CFSMs enabled by that event. If an event carries a value, then it is also assigned an input port. The assumption is made that port holds a valid value at the time the interrupt occurs.

Polled events are assigned one input and one output port (for the acknowledgment), and if necessary additional input ports for the value. The external hardware sets the input port to 1 when event occurs, and resets it to 0 when it is acknowledged. If there are any polled signals, a polling task is automatically generated. This task is run like any other task and can be enabled by a designated event (typically, a timing reference). When it executes, it scans input ports assigned to polled events. If it finds a 1 at one of these ports, the polling task acknowledges it, and calls the appropriate emit function.

MP CFSM → SW CFSM Events from MP CFSMs can also be communicated to SW CFSM by interrupts or polling. In case of interrupts, the only difference is that ISR is not generated, but imported from the library (and parameterized by the actual name of the event that has to be emitted). In case of polling, the difference is that instead of checking ports, the polling tasks calls a library defined function to check if the event has occurred.

3.2 Implementation of coordination

In POLIS, the user can choose between two basic scheduling algorithms to coordinate SW CFSMs: cyclic scheduling and static priority based scheduling. Furthermore, it is possible to choose between preemptive and non-preemptive versions of static priority based scheduling. In either case, RTOS keeps track of events a SW CFSM is sensitive to, and will not execute it unless at east one of those has occurred since the last execution of the SW CFSMs.

Executing a SW CFSMs does not necessarily imply it making a transition. It is possible that the combination of occurred events at the moment of execution does not enable any transition. SW CFSMs provide this information to RTOS so it can decide whether to reset local event flags (in case a transition has been made) or not.

4 Experiments

In order to validate the POLIS generated RTOS we have created a prototyping environment which consists of a complete design flow to generate the final software and hardware elements including boards with hardware components, and instrumentation (e.g. logic analyzer, emulator). This setup allows us to:

- compile, link, and download the software parts,
- program the hardware parts into FPGAs, and
- debug in real-time the software code running on the target architecture.

Both POLIS generated, and commercial RTOSs are being profiled on this platform. The experiments are in progress, and in this paper we present initial results, But first, we give the descriptions of the prototyping environment and the integration of POLIS generated code with a commercial RTOS.

4.1 Prototyping environment

The prototyping environment provides a complete design path from a high-level specification in POLIS to the target architecture. In our case, the target architecture, a typical prototype of an embedded controller, includes the following hardware elements: micro-controller, external memory, FPGAs, and some glue logic (e.g. address decoder). These components are placed and wire-wrapped up on a traditional board.

POLIS takes a specification of the system being designed as an input and outputs a set of automatically generated C programs which implement the SW CFSMs and RTOS, synthesized hardware in XNF format, and connectivity information. C programs are individually compiled and linked together with the user library, and startup code. Commercial design tools are used to place and route and program the FPGA as directed by the POLIS generated XNF code. The connectivity between hardware and software partitions has been reflected in the wiring of the board as well as in the port assignment of the FPGA.[1]

The final binary code is downloaded to the micro-controller through an emulator which offers many debugging and monitoring capabilities. These features allows us to: run the final code in real-time on the target architecture, step it, place break points, measure execution times, ... etc. A logic analyzer is also used for debugging and measuring properties of the system.

4.2 Integration with a commercial RTOS

The POLIS generated RTOS can be substituted by an existing real-time kernel in order to compare the two RTOSs. The integration requires manual modification of the automatically generated C code for SW CFSMs, creation of new I/O tasks, and addition of code for initialization of the commercial RTOS, and proper implementation of POLIS communication model.

The commercial RTOS we are using for this comparison is typical of a commercially available kernels for the micro-controller we use (Motorola 68HC11). It offers over 60 functions that enable the user to create multi-tasking applications for embedded controllers. Task, event, message, resource, timer, queue, and memory management take place through calling those functions. The scheduler is based on preemption, interrupts and other tasks can cause immediate task switch if a task becomes the highest priority task resulting from a system function.

The following changes were done to POLIS generated code to integrate it with the kernel:

- The original POLIS routines for emitting events have been changed to kernel function calls that signal events. To ensure that a SW CFSM is executed only if some relevant event happens it is wrapped with a code that awaits (by a system call) to any event the SW CFSM is sensitive to, sets proper event flags and then executes the POLIS generated SW CFSM code.
- The original polling task, has been re-implemented as an independent task triggered by a periodic event.
- All timer tasks (implemented by MP CFSMs) have been re-implemented by cyclic timer tasks provided by the kernel.
- A new main program initializes the kernel by creating and triggering tasks and cyclic timers.

4.3 Properties to be measured

An industrial example, a dashboard controller, has been chosen for the profiling. The entire specification is implemented in software except interface modules which are in hardware. In the POLIS generated software code the dashboard example has 7 tasks and 4 timers. We are using this example to compare POLIS generated and commercial RTOSs in several areas, namely: task execution, event emission, distribution and detection, I/O operation execution, context switching, control and data code execution times, number of lost events, and code size. Our goal is to use example-specific knowledge to improve over a commercial RTOS in all these areas. We will use experimental results to identify bottlenecks and potential area for further optimization.

[1]Our environment still require some example-specific re-wiring. This could be avoided by using a programmable prototyping board (e.g. APTIX) [1], but so far re-wiring was only a minor portion of total experiment time.

4.4 Results

So far we have experimented with a subset of dashboard specification (a belt controller) that has two tasks and uses two timers, using polling and round-robin scheduling. Initial results are encouraging. For example, if no input events are present, the POLIS generated polling task takes $230\mu s$ compared to $969\mu s$ by the one integrated with the commercial RTOS. Similarly, event emission in the POLIS generated code takes $15\mu s$ compared to more than $150\mu s$ to execute the corresponding system call in the existing RTOS. Code size has also been significantly reduced (from 4.7kb to 1.5kb). By the time of CODES'97 we expect to have a complete set of results.

5 Conclusions

We have proposed to automatically generate RTOS from a higher level description of the system. This approach is faster, less expensive and less error-prone than manual design, yet it allows more example-specific optimizations than commercially available general purpose RTOSs. To validate our approach we have developed a prototyping environment and are using it to compare POLIS generated and commercial RTOSs. Although experiments are still in progress, initial results indicate that POLIS generated RTOS indeed outperforms the commercial one. In the future we plan to complete the experiment and compare POLIS generated RTOS to a manually designed one.

References

[1] S. Cardelli, M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Rapid-prototyping of embedded systems via reprogrammable devices. In *7th IEEE International Workshop on Rapid System Prototyping*, 1996.

[2] 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.