Real Time Operating System Modeling in a System Level Design Environment

Claudio Passerone
Politecnico di Torino - Dipartimento di Elettronica
corso Duca degli Abruzzi 24 - 10129 Torino - Italy
e-mail: claudio.passerone@polito.it

Abstract—Modeling complex embedded systems requires design tools that are able to represent a variety of heterogenous components at several level of abstraction. These components often include hardware and software blocks, which play very different roles with respect to cost, performance and flexibility of the design. Early evaluation of possible design choices and trade-offs is therefore of paramount importance for an efficient implementation. One kind of component that is often neglected in the first stages of the design cycle is the Operating System. However, since it sits between the software and the hardware domains, its effects on the final implementation are relevant.

This paper presents a technique to model a general POSIX compliant Real Time Operating System within the Metropolis framework. Possible models are presented and discussed with respect to their efficiency, taking into consideration the scheduling policy, the multitasking environment, interrupts and communication primitives.

I. INTRODUCTION

Most modern multimedia embedded systems are very complex electronic systems that include many different and heterogenous components. Due to requirements such as short time-to-market and high flexibility, the programmable part is becoming more and more dominant, with application specific hardware used only for very computational intensive and power hungry portions of the design.

Embedded processors start showing performance comparable to those of desktop PCs of a few years ago, and are typically managed using Real Time Operating Systems (RTOS) or versions of general purpose operating systems, such as Linux and Windows, especially tailored for embedded devices [1], [2]. Verification of these embedded systems is a challenge, since full RTL simulation is too slow, and cycle-accurate Instruction Set Simulators require a complete design, including application and RTOS customization, and is thus suitable mostly for final integration and debugging. On the other hand, a functional simulation does not take into account the complex interactions of embedded software with the many peripherals often found aboard these processors, and with the custom hardware units present in the complete product.

Methodologies that are able to work at different levels of abstraction and allow refinements by gradually introducing more details to increase the modeling accuracy are therefore of paramount importance to tackle the verification problem. Although some techniques and frameworks that address these issues are emerging, most of them do not provide sufficient support for operating system and device driver models. As a result, most of the software debugging is performed on evaluation boards or prototypes that carry the real hardware (or a hardware emulation of it), at a time when many key design choices have already been finalised. Thus, early exploration of the design space is not possible, leading to sub-optimal designs from both performance and cost point of view.

In this paper, we address the problem of modeling a subset of a POSIX compliant real time operating system in the Metropolis framework ([3]). We wanted the model to be general enough, so that other kind of operating systems could use the same overall structure and strategy. We also wanted the model to be modular, in order to be able to re-use components in various designs, and extensible, to accommodate different level of details.

II. RELATED WORK

Early attempts to system level design, such as [4], [5], [6], aimed at modeling a function running on an architecture and provided support, in some cases, for the synthesis of some kind of custom operating system. However, they almost completely ignored the timing and performance effects of the RTOS itself, thus resulting in grossly inaccurate estimations, and did not exploit the services of a modern RTOS.

In [7] a software model is annotated with the overhead due to a preemptive scheduler analyzed by running the software tasks on an ISS. While this increases accuracy and is helpful towards the final stages of the design, it is of no help when a designer only needs a rough estimate on his software models, also capturing the operating system overhead.

More recently, to cope with the pressures of time-to-market and design complexity of hardware/software systems, projects like [3] and [8] raised the level of abstraction, and therefore require even more sophisticated techniques to model low level hardware details, as well as more abstract software components. For instance, in [9], [10], the authors propose an RTOS model built over the SpecC language, which provides features typically available in any RTOS to model the dynamic behavior of multi-tasking systems at high abstraction levels. Our approach is similar, but it allow one to more easily model system calls specific of a particular RTOS, and better integrates device driver models and peripherals. [11] is also similar, but requires its own proprietary simulation engine. In [12] the authors propose a framework based on SystemC to model RTOSs in a multiprocessor environment: our approach is more detailed and can be extended to multiprocessor systems as well, within the Metropolis framework.

III. THE METROPOLIS FRAMEWORK

Metropolis ([3]) is a system-level design infrastructure based on a model with precise semantics that is general enough to support existing computation models and accommodate new ones. To allow better reuse of models, the focus is on separation of independent aspects, such as computation
and communication, functionality and architecture, behavior and performance. The system to be developed is hierarchically decomposed into several objects to define functions, architectures and mapping. Special objects, called quantity managers, are used to drive the execution of the system and assign cost and performance.

Function is defined by a network of concurrent processes that communicate through communication media. Processes and media are both described using a sequential language called metamodel. The behavior of the network is a sequence of event vectors, where each event represents the beginning or the ending of a computation or communication step by a particular process. The architecture provides computation and communication services to the function. It is a network of processes and media as well, and services are specified using the same metamodel used in the function side. Services are decomposed into a sequence of events; each event can be annotated with a value representing its cost and performance, in terms of a number of physical and abstract quantities.

Mapping is used to associate functions to services provided by the architecture, by synchronizing events occurring within function processes with events occurring in the architecture.

Simulation of a design can be divided into two phases, the request phase and the resolve phase. In the request phase, software tasks and hardware-mapped processes execute their behavioral description till they reach a point where a request for performance annotation is made: they will generate an event and queue in their annotation request to the quantity managers which control the physical quantity that they want to be annotated with. After all the tasks have made such requests the resolve phase starts: based on its policies and the set of annotation requests from the tasks, the quantity manager will annotate the selected requests and notify the owners. The request phase now starts again. The notified tasks will further execute their behavioral descriptions till they reach the next annotation point.

IV. MODELING AN RTOS

A Real Time Operating System is a complex software infrastructure that is made of several components. It presents a virtualised processor to a set of software tasks, and manages the hardware resources of the CPU to best exploit them. A simple RTOS can be schematically represented as in Figure 1, where the various parts that constitute it and the interactions with other components of the system are shown.

We considered several modeling alternatives, in which the various components, and in particular the Kernel API and the Scheduler were separated into distinct blocks. The scheduler was modeled as a quantity manager, connected to the CPU on which the RTOS is running and controlling all user processes that need to be scheduled on that CPU. Regarding the kernel, we studied two options:

- Implemented as a privileged process running on the CPU.

The process is normally idle, and only wakes up when a process requests an API to be run.

- Implemented as a communication medium connected to the CPU, but not to the user processes. The CPU collects traps from the user processes requesting services to the kernel, and forward them to the communication medium modeling the kernel.

Although both solutions are possible, we chose the second one because it is easier to implement, and showed better performance in simulation, since synchronization is achieved through simple function calls rather than through a scheduling mechanism. Therefore, in the following, we will refer to this architecture only.

During normal operation, when a user process is running and no RTOS activity is needed, processes simply issue instruction execution requests to the CPU, which returns the number of clock cycles consumed. However, when a process needs a service from the OS, it will issue a special trap request, with the specific API function specified as a parameter. Then the CPU, whose model must implement the trap interface, calls the kernel and accounts for its activity, which depends on the function that is being executed.

When a user process goes to sleep, either because it is waiting for the CPU or another architectural element to process a request, or because it voluntarily suspends using a system call, the RTOS scheduler is invoked and a (possibly) new user process is selected to run on the CPU. The information needed for scheduling are shared with the Kernel API, and several different policies can be implemented.

A. RTOS features

We planned to implement an RTOS that would be as close as possible to a POSIX compliant operating system. Based on this decision, we selected the features that had to be supported by our model. Since it is a multi-tasking environment, we needed to implement preemptive control of user processes, semaphores for synchronization and inter-process messages for communication.

A partial list of the system calls that were implemented in our experiments is shown in Table I. Some of the system calls not shown in the table are related to error conditions, such as arithmetic errors or segmentation faults, that can be issued by software tasks to kill the offending process, for instance. We only specified a priority based scheduling mechanism, although other policies such as Round Robin or Rate Monotonic can be easily added. A time-sliced model has also been developed, and was presented in [13].

Another critical feature that we wanted to support is the ability to respond to interrupts from external hardware devices, and the possibility to control peripherals using drivers. The physical device is modeled as a process in Metropolis, since it often has some autonomous behavior that runs independently of the CPU and the software domain. The control of the device is achieved through a communication medium that is used in two directions:

1) From the CPU to the physical device, to write and read the device configuration registers and issue commands to the peripheral.

2) From the physical device to the CPU to raise interrupts. Interrupt service routines and drivers are contained in a communication medium within the RTOS, that implements all
Process control

- getpid: returns the process identifier of the running process.
- getpriority: returns the priority of the running process.
- ch.priority: changes the priority of the specified process and returns the old priority.
- sleep: puts a process to sleep for the specified time.
- suspend: suspends the specified process.
- resume: resumes the specified process.

Semaphores

- sinit: creates a new semaphore and initializes it.
- destroy: deletes the specified semaphore.
- getvalue: returns the counter value of the specified semaphore.
- wait: blocks on the specified semaphore.
- strywait: non-blocking P on the specified semaphore.
- s.post: V on the specified semaphore.

Messages

- msend: sends a message to the specified process.
- mrecv: gets the next available message.
- m.tryrecv: gets the next available message (non-blocking).
- mbclear: clears the mailbox of the calling process.
- mbgetn: returns the number of waiting messages in the mailbox.

The hardware and software vectors, in a way similar to how the Kernel API system calls are modeled. Multiple device can be instantiated at the same time, thus allowing to model very complex CPU architectures. System calls within the Kernel API are used by processes to enable and disable interrupts, using a masking mechanism.

To raise an interrupt, a physical device issues a trap-like signal with an interrupt id to the CPU, which in turn calls the interrupt service routine specified in the vector handlers and triggers the scheduler. The routine is allowed to run only if the priority for the interrupt is higher than the currently executing task. When the routine terminates, another scheduling step resumes normal operation.

Examples of peripherals that were modeled are a DMA device, a timer and a serial port. In all cases they used both communication with user processes (to set the starting address and number of byte for a DMA transfer, or to set a timeout, for instance), and interrupts to the CPU (to signal the end of a DMA transfer, or the arrival of new data over the serial port). After adding all the features that we wanted, the final architectural netlist that is obtained is illustrated in Figure 2.

B. Simulations

We performed several simulations to test the functionality of the RTOS. The simplest simulations involved a set of statically created tasks which randomly executes instructions on the CPU and invokes system calls to change process priorities or sleep. More complex simulations, on the other hand, show the interactions between task scheduling and some peripherals managed using device drivers and interrupts.

Time is modeled by inserting appropriate time requests to a global time quantity manager. These requests typically come from the CPU model or other architectural elements like a bus controller or hardware devices. The CPU itself receives computation requests from the tasks, as well as from the operating system model, and transforms them into time, based on the amount and type of code to be executed, the available resources and the simulated clock frequency. The granularity and accuracy of the estimated annotations determines how detailed and precise the simulation results will be: obviously designers have to trade-off simulation speed versus accuracy.

Figure 3 shows code extracted from a system where a process starts some device action, by writing a particular value on a specific register, and then wait on a semaphore the completion of that action. In the code, did is the device identifier, reg is the device register to be written and val is the new value; sid is the semaphore identifier, which is created by the task itself using a system call. Both writing to the device register and managing the semaphore through system calls is achieved using a port connected to the CPU medium. The device normally waits (using Metropolis metamodel communication primitives) for registers to be written, and checks their value; it will perform some computation, and then will issue an interrupt, whose handler is responsible for resuming the process that is waiting on the semaphore. Communication from the device to the CPU, the interrupt handler and the scheduler always happens through the device register medium.

In this case, to make the example simpler, the device registers were written directly by a user task. However, the same action can be performed within a kernel API. If the above system simulates a DMA transfer, a task would typically issue a system call to specify the starting memory address and count, and the implementation of the system call would write the appropriate registers, thus implementing a device driver for that specific architecture.

The complete system also includes other processes with lower priority with respect to the one that access the hardware device. When simulating this system, we can plot a chart showing the time spent in each task, as shown in Figure 4. In the figure, we distinguished the low priority tasks (tasks 1, . . . , n), the task that communicates with the hardware device (task 0), the device itself and the kernel APIs and interrupt vectors. The high priority task gains exclusive use of the processor while it is running, but other processes are allowed to run while the device is active and task 0 is idle, waiting on the semaphore for completion of the hardware.

The performance of the simulation measured in clock cycles per seconds of simulation depends on the level of details in the user tasks and kernel models. In our case we used a coarse grain of annotation, corresponding to time synchronizing only at communication points between processes. Hence performance was almost like real time (thanks to the fact that simulation runs on a powerful workstation).

In a more complex system, the hardware device also accesses shared resources, such as a bus and a memory, that are also used by the other processes running on the processor. Assuming that a DMA transfer is going on, and that the DMA device has a higher priority on the bus, then depending on the relative timing, it is possible that tasks running on
models. Modern embedded systems are composed of a variety of hardware and software components, and often include an operating system to virtualize the hardware platform and present a general and common interface to the software tasks. In this paper, we described a technique to model a subset of a POSIX compliant real time operating system in a system level design framework. The model, while taking into account custom hardware peripherals that are accessed by the software domain, only requires to implement a standard interface to call software traps and react to interrupts. This allows better re-use of the architecture model in general, and of theRTOS model in particular.

In our experiment, we have shown how to verify that the behavior and the performance of a design involving a real time operating system meet the required constraints. In particular, we focused on the complex interaction between hardware peripherals and software device drivers, that are today still verified on prototype boards. Our technique allows to move part of this verification step earlier in the design flow, thus decreasing the number of design iterations and ultimately leading to a shorter time-to-market and better overall results.

REFERENCES


Fig. 4. Simulation example with SW/HW interaction

V. CONCLUSIONS

System Level Design facilitates modeling at high-level of abstraction as well as mixing highly detailed and low detailed

process Task {
  port UserCpuIntf cpuPort;
  public void thread() {
    int did = 1, reg = 0x04, val = 0x0001;
    sema = new pSemi();
    /* sem_init(&sema, 0, 0); */
    cpuPort.swTrap(new SysCall(SINT, pid, 0));
    /* Implement process behavior */
    ...
    cpuPort.deviceCommand(did, reg, val);
    /* sem_wait(&sema); */
    cpuPort.swTrap(new SysCall(SWAIT, pSema, 0));
  }
}

medium Cpu implements UserCpuIntf, DevCpuIntf {
  port CpuKernelIntf kernel;
  port CpuVectorsIntf vectors;
  port CpuRegsIntf device;
  public int swTrap(SysCall sc) {
    return (kernel.IntrinsicInsert((sc));
  }
  public int deviceCommand(int did, int reg, int val) {
    device[did].putReg(reg, val);
    return 0;
  }
  public void sendInterrupt(int from) {
    scheduleInterrupt(from);
  }
  public int intManager(Intf intr) {
    vectors.intManager((intr);
  }
}

process Device {
  port DeviceRegsIntf regPort;
  public void thread() {
    while (1) {
      int tv = regPort.getReg(0x04);
      if (tv & 0x01) {
        /* Do something */
        ... regPort.putReg(0x04, regPort.getReg(0x04) & 0xfo);
        regPort.sendInterrupt(pid);
        regPort.intManager((intr);
      }
    }
}

medium Vectors implements CpuVectorsIntf {
  port KernelTableIntf sysTable;
  public int intManager(Intf intr) {
    switch (intr) {
      case 0: handler0(); break;
      case 1: handler1(); break;
      ...
    }
    private int handler1() {
      semaphore s = sysTable.getsem(0);
      if (s.queue.size() > 0) {
        tmp = (process) s.queue.remove(0);
        sysTable.chState(tmp.pid, 1);
        sysTable.adtReady(tmp.pid);
      }
    }

Fig. 3. Code example for SW/HW interaction

the processor are interrupted if they try to access the bus themselves concurrently with the DMA device.

Other interesting aspects concerning the real time behavior of a system can also be analyzed. For instance, the time needed to react to a particular event (e.g., an interrupt) can be studied. A device with its own thread would generate a periodic interrupt, and the interrupt service routine shall set a specific register of a device when finished. If, before issuing the next interrupt, the register is not set, then a notice to designers can be generated, signaling that a deadline has not been met.