Scheduling Issues in the Co-Synthesis of Reactive Real-Time Systems

Pai Chou, Elizabeth A. Walkup, Gaetano Borriello
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

April 20, 1994

Abstract

Many embedded control applications must respect intricate timing requirements on their interactions with the external environment. These constraints are derived from response time, rate of execution, and low-level signaling requirements. Currently, most of these systems are being designed in an *ad hoc* manner. Many tools assume the designer has already finalized the scheduling, while most schedulers make simplifying assumptions and often cannot handle general timing constraints.

In this paper, we discuss the scheduling issues that must be addressed by co-synthesis tools for embedded systems and outline possible approaches to the problems. Our perspective is based on experience with Chinook, a hardware-software co-synthesis system for reactive real-time systems, currently under development at the University of Washington. Chinook is initially targeting embedded applications without operating system support. From a high-level specification and a device library, Chinook synthesizes both interface hardware and a software program to realize the design.

1 Introduction

Embedded computers that are characterized by their continuous interaction with the environment are called *reactive systems.* Examples of reactive systems range from personal electronics such as digital watches to automotive and jet engine control. When the correctness of such a system is defined by both its *logical* and *timing* behavior, it is classified as a reactive *real-time* system.

To enable rapid exploration of the design space, designers of reactive systems often use off-the-shelf, reprogrammable components and various peripheral devices. Such an architecture consists of one or more microprocessors running software programs to control a number of *devices* which interact with the environment. (See Figure 1.) Examples of devices include temperature and

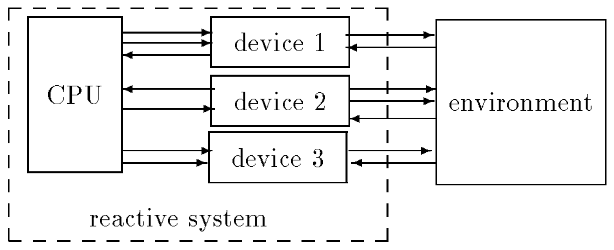


Figure 1: Reactive System Architecture

motion sensors, liquid crystal displays (LCDs), keypads, buzzers, motors that control robot arms or wheels, and bus controllers.

Reactive real-time systems must respect intricate timing requirements at different levels. First, for the microprocessor to communicate with a device in the system, it must generate a sequence of low-level control signals and read or write I/O pins within appropriate time intervals. This information is usually found in the databook for the device. Secondly, there may be more timing constraints defined at a higher level. For example, the specification may require that a button device be sampled once every 20 ms; or it may require that we "stop the motor between 50 and 100 ms after the button is pressed." These are referred to as *rate* and *response time* constraints, respectively.

Scheduling is an error-prone process that requires computer assistance to consider the many interactions between constraints. Unfortunately, current design practices for reactive real-time systems are *ad hoc* and not very retargetable. Programmers meet timing constraints by tuning their code to a specific processor with a particular I/O configuration. Such practices result in poor modularity and limited retargetability, thus severely discourage exploration of the design space. This is the case even if the program is written in a high-level language.

Designers have used *real-time kernels* to solve some of these scheduling problems. Most scheduling work in real-time systems assumes a *process-based* model, where a set of coarse-grained *processes* are scheduled by a real-time operating system. Timing constraints are specified in terms of the *release time* (earliest allowed start time), the *deadline*, and the *period* of the process. However, the fundamental problem with this model is that it schedules processes, while timing constraints are more naturally specified between *observable events*.

In the process-based model, each process may generate a number of observable events during its execution. The scheduler is responsible only for dispatching the processes at the right times, and has no direct control over when the observable events are actually generated by the processes. In practice, designers often must tune their programs manually even when a real-time kernel is used. A few recent techniques such as that of Gerber and Hong [1] attempt to gain better control over observable events by applying compiler analysis. However, they do not handle all important classes of deterministic timing constraints.

Process-based real-time models often make assumptions that may not be reasonable for reactive systems. First, many schedulers rely on preemption and context switching at arbitrary points. Since reactive systems are I/O intensive by definition, this assumption is not reasonable, because an I/O protocol should not be left in an inconsistent state (in the middle of a bus write, for example) by preemption. On the other hand, it is not practical to enclose all I/O protocols in critical regions. Critical regions achieve atomicity by preventing interleaving, and are designed for short accesses to shared data structures, but they are not suitable for those protocols with long separations between consecutive events. Interleaving concurrent I/O transactions is often necessary for reactive systems.

Reactive systems require a different programming model from the process-based real-time model. Among the most widely known specification languages for reactive systems are Esterel [2] and StateCharts [3]. Both provide constructs for concurrency and watchdog-style preemption. Both also define simulation semantics for real-time behavior on an idealized machine, (*i.e.*, one that is "much faster" than the speed of its environment) but of course there is no guarantee that this will be the case. Neither model allows timing *constraints* to be specified, and both assume real-time scheduling is done in advance by the designer.

In Chinook, we extend the reactive programming model by defining a taxonomy of timing constraints, and develop scheduling algorithms to satisfy these constraints while maintaining the integrity of the I/O protocols. We divide the scheduling problems into two levels: system behavior at the high level (Section 4) and device drivers at the low level (Section 5).

2 The Chinook Co-Synthesis System

Input to Chinook consists of a high-level description of the system and a library of processor and device specifications. Chinook synthesizes the software for the microprocessors and any required glue logic. The synthesis steps include software-hardware partitioning, device driver synthesis and low-level scheduling, I/O port allocation and interface synthesis, system-level scheduling, and code generation. An overview of the Chinook system is given in Figure 2. In addition, simulation can be performed on the results from various synthesis stages. In this paper, we focus on only two parts that address scheduling issues: system-level scheduling and device-driver synthesis.

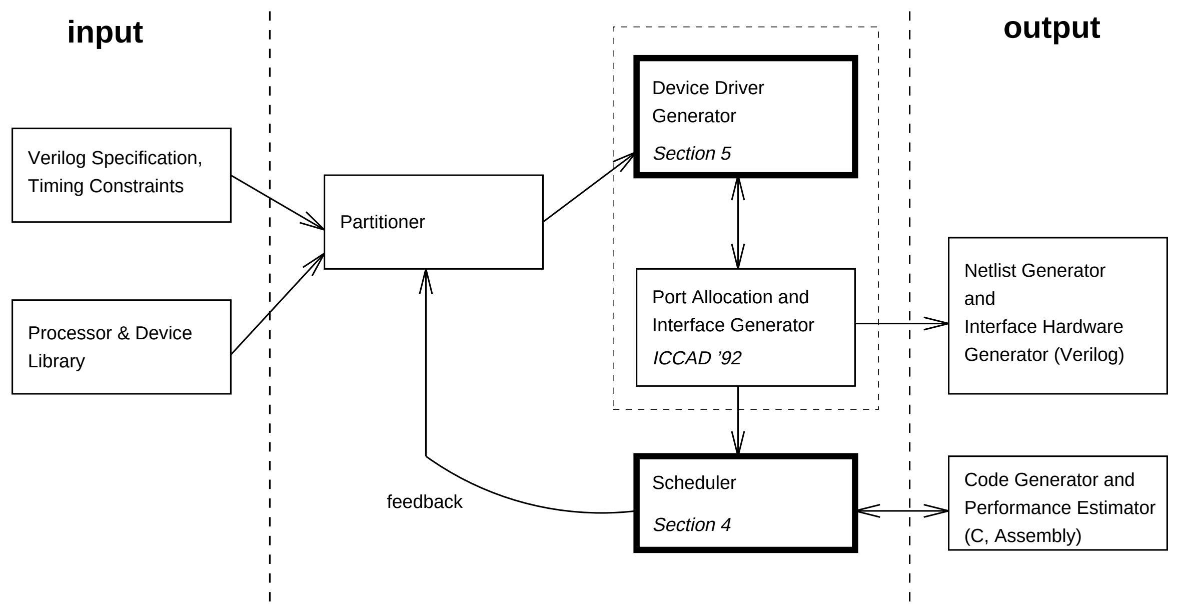

input
output
Verilog Specification, Timing Constraints
Processor & Device Library
Partitioner
Device Driver Generator
Section 5
Port Allocation and Interface Generator
ICCAD '92
Scheduler
Section 4
feedback
Netlist Generator and Interface Hardware Generator (Verilog)
Code Generator and Performance Estimator (C, Assembly)

Figure 2: Chinook System Overview.

2.1 High-Level Specification

The input supplied by the user is a high-level description of the system. Currently, we support the Verilog hardware description language [4]. It contains a structural section and a behavioral section.

The structural description instantiates the processor(s) and the devices used in the system. Chinook automatically connects the pins of the devices to the processor, by synthesizing appropriate glue logic as necessary. The designer may also choose to bypass the automatic allocation mechanism by explicitly specifying interconnections.

The behavioral section describes the functionality of the system, using high-level language constructs such as conditionals, loops, procedure and function calls, and arithmetic and logical operators. For reactive modeling, it supports parallelism, **disable**-style preemption (forcing a block of statements to terminate), real-time delays, and communication through named events. High-level timing constraints can be specified as annotations on the behavioral description. These are described in more detail in Section 3.

2.2 Processor and Device Libraries

The libraries for processors and devices contain information needed to connect them to each other, both physically and logically. For a processor, we define the collections of pins that form parallel I/O ports, serial ports, address/data ports, and their relations when they share the same pins. In addition, we also list the processor's I/O instructions with their timing. For a device, we define its ports, the protocols for accessing the device, the functionality of the device in case it should need to be synthesized from scratch, and a procedural interface. The designer uses this interface in the behavioral specification, thus being shielded from the details of the interactions with the device. An example procedure would be a `bus-write` for a bus interface, taking address and data as arguments. The procedure encapsulates the details of bus arbitration and signaling on the bus. The low-level timing constraints are specified as part of these procedures.

2.3 Partitioning and Device Driver Synthesis

The first synthesis step partitions functionality between hardware and software implementation, and among the processors. The default partitioning is to implement all of the structural device instantiations in hardware and turn all the behavioral statements into software. However, based on the processor selected and both the system-level and device-level timing requirements, we can detect that certain constraints cannot be met if implemented in software. If so, these functions are pushed into the hardware partition. Chinook then synthesizes the device and its interface hardware as a new devices with updated driver routines. This combined partitioning and synthesis problem is addressed in Section 5. Partitioning may be revisited if the high-level scheduler discovers that it cannot meet a performance requirement.

2.4 I/O Port Allocation and Interface Synthesis

Processors need to be connected to the devices they control. If a processor has general-purpose I/O ports, Chinook will attempt to use them because they incur minimal hardware and software cost. A greedy algorithm [5] attempts to use the same I/O port to service those devices that are not active at the same time. If there are not enough I/O ports to service all the devices, then multiplexing hardware is synthesized along with the driver routines. If the processor does not have general purpose I/O ports, then a memory-mapped scheme is used to access these devices. Under user guidance, Chinook assigns addresses to the devices, and synthesizes the address-decoding logic. All device driver procedures are updated to reflect the binding to I/O ports or the use of multiplexing logic.

2.5 System-Level Scheduler

After the resource binding is completed, we can more accurately estimate the run time of the operations and perform scheduling. Scheduling is required to serialize the initial behavioral specification, which may contain concurrent elements. In Section 4, we discuss methods for satisfying sequencing, rate, and response time constraints. If no feasible schedule exists, then we must consider re-partitioning the design among multiple processors or moving functions into hardware.

2.6 Output

The output of Chinook provides all the elements needed to construct the complete embedded system. The principal parts are the net-list of components (devices, processors, and synthesized glue logic) and the code for each processor in the system. For the code generation task, we use a retargetable compiler that also provides estimates of code execution time and code size. These estimates are used during the partitioning and scheduling steps.

3 A Model for Reactive Real-Time Systems

Our model for reactive real-time systems captures reactive behavior with control constructs similar to those in Esterel and StateCharts, but also allows the specification of general deterministic timing constraints between I/O operations to be scheduled. Scheduling is divided into two levels. At the low level, the scheduler/partitioner (Section 5) schedules those operations with constraints on or below the order of the CPU instruction cycle time, as meeting these constraints may require both hardware and software. Each group of operations that have been scheduled together at the low level appears as a single atomic sequence of software instructions to be scheduled at the high level (Section 4).

3.1 Reactive Control Flow

Reactive behavior can often be conveniently and succinctly expressed with concurrency and watchdog-like preemption, two of the common features of Esterel and StateCharts. We have chosen structured control flow as in Esterel, instead of arbitrary state transitions with go-to semantics as in StateCharts.

Our model uses **fork-join** and **disable** primitives, similar to those in Verilog. A **fork** specifies a list of concurrent operations. A corresponding **join** waits for the forked operations to terminate before passing control to the next statement. A **disable** causes a named block of statements to terminate. A disable in conjunction with a fork can be used as a watchdog. For example, one branch of a fork can be the main loop of the system, while the other branch watches the reset button until it is pressed and then disables the entire fork statement block. Watchdogs may also be used to describe timeout behavior.

One immediate question is “where does the disable take effect?” In Esterel, there are two possibilities under the *perfect synchrony hypothesis*, which states that operations consume no time to execute. The first possibility is for the watchdog to terminate the code being watched at the beginning of an instant (Esterel’s **watching** construct). The second possibility is to give its peer statements a chance to run until they block and then terminate them (Esterel’s **trap** construct).

Our disable is similar to the **trap** construct in Esterel, although the timing semantics is different because operations in our model do consume time. To ensure that I/O's and operations in general are not disabled in an inconsistent state, we allow the disabled operation to define a number of *safe-exit points*, where disables may take effect. This is especially important for maintaining the integrity of I/O interactions while enabling interleaving. At the high level, the safe-exit points can be used to give the disabled operation a chance to save its state.

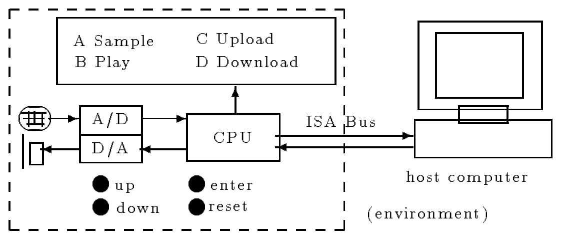
3.2 Timing Constraints and Modes in Reactive Systems

Timing constraints are the minimum or maximum separations between pairs of events. In hardware, events include rising and falling edges where timing constraints such as set-up and hold times are defined between edges. In software, we define the events to be the *start* of a software operation. Since we use watchdogs for modeling reactive behavior, each watchdog defines a natural scope in which a set of timing constraints is active at the same time. Each such scope is called a *mode*. Thus modes are quite similar to a hierarchical state in StateCharts, but also include the timing constraints that must be satisfied in that state.

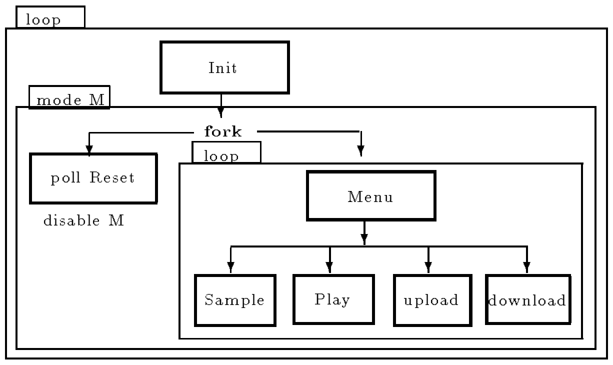
We divide the types of timing constraints into those within a mode and those between two modes. Timing constraints within a mode include *sequencing* and *rate* requirements. A sequencing constraint is the minimum or maximum separation between the start of two events in the same iteration. A rate constraint is the min/max separation between the start of consecutive iterations. When a watchdog detects an event, it disables the mode and causes a mode transition. Timing constraints can also be specified between two operations across a mode transition. Such an intermodal constraint is known as a *response time* constraint.

3.3 An Example

To help illustrate the concepts in this paper, we consider a simple speech sampling and playback system (Fig.3). The system consists of a CPU, an LCD, four buttons (up, down, enter, reset), an analog-to-digital converter (ADC) connected to a microphone, a digital-to-analog converter (DAC) connected to a speaker, and an interface to the ISA bus. Chinook's initial partitioning is to



System Architecture



Top-Level Modes

Figure 3: Example Speech Capture/Playback System with Bus Interface

instantiate only the required hardware (LCD, buttons, ADC, and DAC), and implement as much as possible in software, including the bus interface. On startup, the system enters the initialization mode first, and then enters the menu mode to poll the buttons and update the menu on the LCD. The menu selection will cause the system to enter one of the following modes:

1. Record sound by reading 8-bit sound samples from an A/D converter at 20KHz and store them in a 16K memory.
2. Play back the sound in the buffer by writing to the D/A converter at 20KHz. In addition, pressing the UP button during recording causes a time stamp to be recorded.
3. Upload the sound data to a host computer via the ISA bus
4. Download the sound data from the host computer.

If, at any time, the reset button is pressed, the system restarts from the initialization mode by disabling mode M, which contains the main loop and the watchdog for reset in parallel.

The timing constraints at this level include rates and response times. An example of a response time constraint is "start sampling within 1 ms after the enter button is pressed." In addition to the 20KHz sampling and playback rate, the buttons may also have a polling rate constraint.

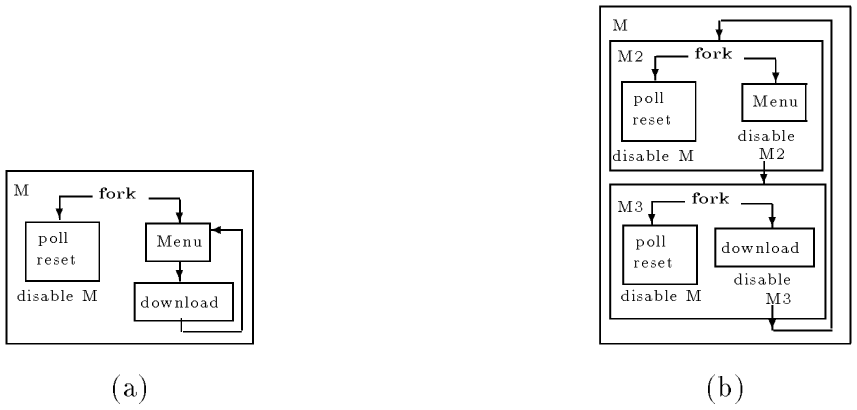


Figure 4: Example of Flattening

4 Scheduling in the Reactive Real-Time System Model

The system-level scheduling problem can be classified as fine-grained, static, and nonpreemptive. Operations may have nonuniform integral worst-case execution times and are related by precedence (*i.e.*, partial order) defined by the minimum and maximum separations between pairs of operations.

Instead of computing a single static schedule for the entire reactive system, our approach is to produce multiple schedules, one for each mode of the system. When the system changes mode, it starts running with the new schedule for the new mode. In the general case, modes are hierarchical and may contain parallel loops with different rate requirements. These mode structures require transformation before they can be scheduled statically. The following two subsections discuss how to flatten hierarchical modes and parallel loops so that they can be scheduled statically to meet the intramodal and intermodal constraints.

4.1 Scheduling Hierarchical Modes

Hierarchical modes must be *flattened* before they can be scheduled. If a watchdog watches an event over multiple mutually exclusive modes within a hierarchical mode, the watchdog must be "inlined" for each of these nested modes to flatten the hierarchy. In addition, if the watchdog has rate constraints, we may need to add them as intermodal constraints.

Consider the speech sampler example in Figure 4(a) prior to flattening. The watchdog for the reset button is active when the system is in either the menu mode or the download mode. To flatten the hierarchy, the reset watchdog is replicated in both nested modes, as shown in Fig. 4(b). If the reset watchdog has a rate constraint, then an intermodal constraint equal to its period is added from the watchdog in M2 to the one in M3, as well as from M3 to M2. If either the menu mode or the download mode is hierarchical, then flattening continues recursively. Flattening results in parallel loops that can be processed by the following techniques.

4.2 Scheduling Parallel Loops

Here we consider the case where a mode consists of several loops running in parallel. Each loop may have some minimum and maximum rate constraint, and the body of each loop can have sequencing constraints. An example of this structure is when the system needs to service or watch inputs from several devices in parallel, each with its own protocol and rate requirement. We can schedule this mode by first transforming the parallel loops into an outer loop with a parallel body, so that the body of the loop can be scheduled using known techniques [6]. Here we describe the transformation step called *rate matching*.

The new loop must have a rate constraint such that the bodies of parallel loops can be executed at their required rates. If they have different rates, then the new loop period is the least common multiple (LCM) of the periods. A loop L_i with a period p_i is unrolled $\mathrm{LCM}(p_1, \cdots, p_n)/p_i$ times. Several techniques can help reduce this potential code explosion: exploiting the freedom in the rate requirement, performing *partial unrolling*, and converting a polling loop into an interrupt.

Often the rate constraints are specified as ranges rather than just a single value. We exploit this freedom to minimize the unrolling factor, especially when either the upper or lower bound is left unspecified. In the speech sampler example, the "playback" mode contains three parallel loops: polling the reset button, polling the up button, and playing the sound. The button polling loops have a maximum period constraint of 10ms but no minimum period constraints; the playback loop has a required period of 50μs (Fig. 5(a)). By matching the button polling rates with the playback rate, we can completely eliminate the need for unrolling (Fig. 5(b)).

Instead of matching the periods, *partial unrolling* unrolls a faster loop enough times to match

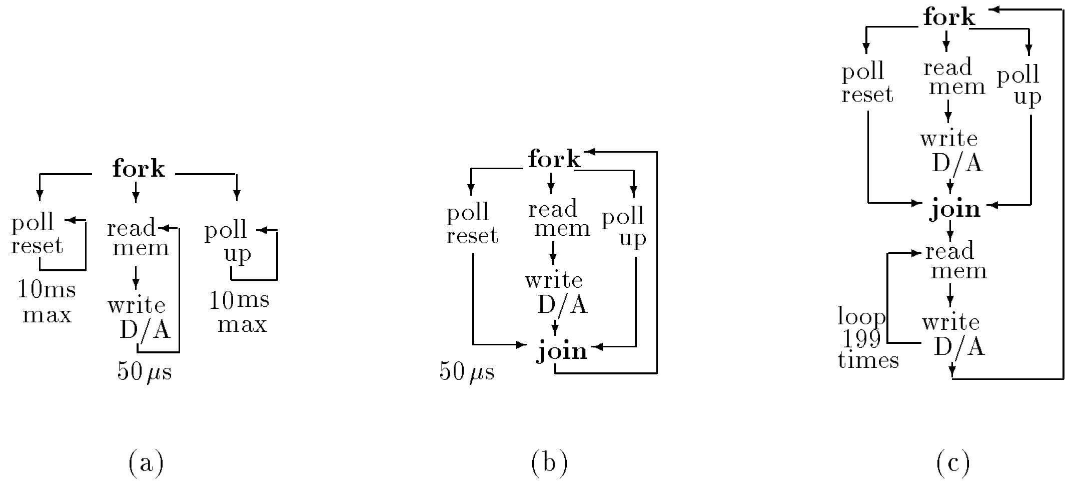


Figure 5: Example of Rate Matching (b) and Partial Unrolling (c) of the structure (a).

the *execution latency* of the slower loop. To illustrate this method, assume the polling loops have a minimum period constraint of 10ms also. Since the execution latency of the button polling loops is short, partial unrolling eliminates unrolling the playback code during the idle time of the polling loops (Fig. 5(c)).

Sometimes it is desirable to transform one or more watchdogs from polling loops into an interrupt implementation. Such watchdogs can be characterized by having a fast polling rate relative to their peers, a short response time constraint, and/or a response action requiring little computation time. These watchdogs also tend to watch over a large hierarchical mode, such as the reset watchdog in the speech sampler example. Extracting the code from the fork and implementing it with an interrupt not only saves code size but also reduces the processor load. Information regarding the frequency of interrupts must also be known to ensure that other loops are not adversely affected by this transformation. This type of transformation requires further investigation.

4.3 The Core Scheduling Algorithm

The core of the scheduling problem serializes concurrent operations for each mode and assigns their start times to meet timing constraints. To compute a schedule within a mode, we assume that the mode has been transformed using the techniques already discussed, so that it contains no loops. The scheduling problem can be formulated as a graph, where the vertices represent operations, and the edges represent timing constraints similar to relative scheduling [7]. Our scheduling algorithm [6] calls the Bellman-Ford single-source longest path algorithm as a subroutine both to check feasibility and to assign start times of the operations. To take inter-modal constraints into account, we schedule each mode with incoming and outgoing constraints on the mode transitions.

5 Automatic Device Driver Synthesis

In addition to satisfying the real time constraints specified for our application, we must also satisfy the timing requirements of all peripheral devices with which the microcontroller communicates. We do this by encapsulating device behavior within *device drivers*. These drivers consist of a script of software instructions for the microcontroller and interface hardware between the microcontroller and devices. The device drivers are responsible for generating the appropriate signal sequences to interact with each device correctly. The higher-level software, which implements system functionality, can then invoke the drivers without attending to the hardware and timing details of the peripheral devices. Thus, device drivers create a useful layer of abstraction between hardware requirements and system performance requirements while permitting optimization within the device drivers themselves. The problem of determining a good implementation of the driver – so that it meets timing constraints imposed by the device and makes efficient use of hardware resources – can be posed as a combined scheduling and partitioning problem.

Previous work in the field of interface synthesis [8] considered the problem of generating glue logic to interconnect devices whose interfaces are specified by timing diagrams. Given the presence of a microprocessor in the systems we are considering, it is natural to implement much of that interface logic as software routines and thus reduce the cost of interface hardware while providing added flexibility. However, interface hardware may still be necessary, even with today's microprocessors and microcontrollers, for performance and bandwidth reasons. The processor may not be fast enough to meet the timing constraints of the devices; the processor may not be able to achieve the interface throughput required; or the processor may not have enough external pins (ports) to directly connect with all the devices that it must control. This paper addresses the *performance* problem; bandwidth issues of port allocation and memory mapping are handled by the algorithm described in Chou *et al* [5].

Different processors with different speeds, instruction sets, and I/O ports will require different variations on the software routines. In addition, though it is possible to write a standard suite of device driver routines for each device-processor combination, there are several reasons it may be advantageous to synthesize new drivers automatically. For example, an application which uses only a subset of a device's possible operations may not require as much interface hardware as one using a comprehensive set. Furthermore, even when the microprocessor speed is well matched to the device's communication requirements, tighter system-level real-time constraints may overload the microprocessor and thus force more system functionality to be implemented in hardware.

When possible, it is desirable to separately satisfy the timing constraints induced by the devices and those induced by the real-time constraints, since the latter will tend to involve time intervals an order of magnitude larger than the former. With such issues in mind, we wish to create for each device a *driver* consisting of software routines and interface hardware connecting the microprocessor to the device, if necessary. The software routine implements an atomic device operation either directly or indirectly by driving the interface hardware to meet the device's timing constraints and realize the transaction.

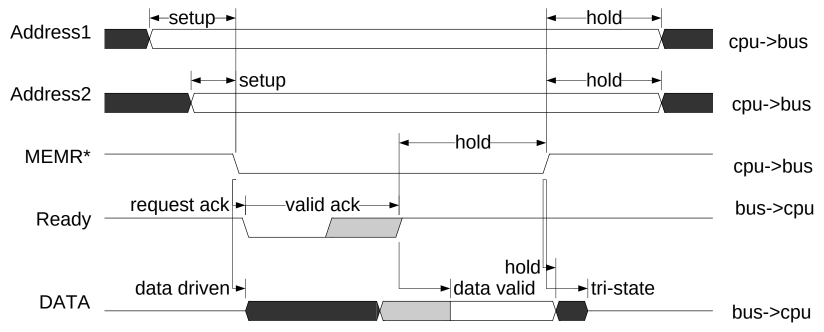


Figure 6: Timing diagram for ISA bus `ready-read` operation.

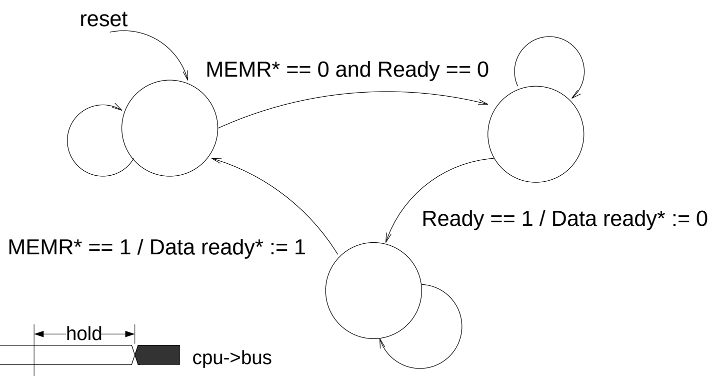
Device behavior is generally described with timing diagrams. Figure 6 shows an example of the ISA bus `ready-read` operation. In this example, a simple software device driver would read and write I/O ports directly connected to the bus and execute the following steps:

- provide data for Address1 and Address2
- drive MEMR* low

- poll Ready until its value is 0
- poll Ready until its value is 1
- read a word from DATA line
- drive MEMR* high

Suppose, however, that the amount of time between the events *request-ack* and *valid-ack* on the Ready line is small enough that the microprocessor we have chosen cannot be guaranteed to catch the zero value on the Ready signal line. If this event were missed, the microcontroller would have to assume that the read command it issued was not received by the bus. To solve this problem, we can introduce a hardware finite state machine to watch the MEMR* and Ready lines and to catch the *valid-ack* event. This state machine generates a signal (Data-ready*) that can be polled by the microcontroller at a later time to determine if the data are valid. Figure 7 shows a possible device driver for this scenario. It consists of microcontroller code, a hardware finite state machine, and a timing diagram which depicts the new timing relationships between signals.

```
ISA Read(in adr1,in adr2, out dataReg):
        Address1 := adr1;
        Address2 := adr2;
        MEMR* := 0;
        While(Data ready* == 1);
        dataReg := DATA ;
        MEMR* := 1;
```


reset
MEMR* == 0 and Ready == 0
Ready == 1 / Data ready* := 0
MEMR* == 1 / Data ready* := 1

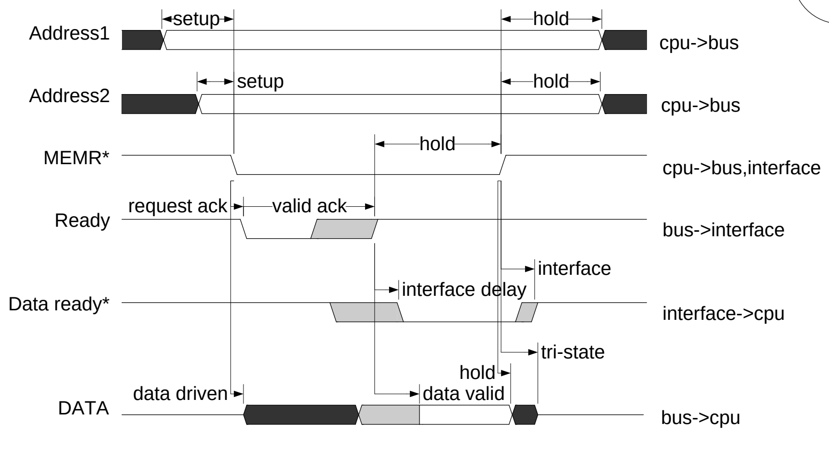


Figure 7: Device driver for ISA read operation, consisting of a microprocessor routine and hardware finite state machine.

5.1 A Combined Scheduling and Partitioning Problem

Events represented in the timing diagram fall into two categories: *device events*, which are generated by the device itself, and *driver events*, which must be generated by any user of the device. For the Ready signal in Figure 6, the device events are *request-ack* and *data-valid*. In addition, there are two implied driver events, namely reading a zero value on the Ready line, and then reading a one value.

Creation of the device driver consists of three steps. First, for each device routine used, we partition the driver events into two sets, those that can be controlled directly by software and those that necessitate external hardware. This step will introduce interface signals for controlling the interactions between the software and the external hardware. Second, for each different device call, we schedule a software subroutine consisting of those driver events we can control using microprocessor I/O instructions. Third, we interface the processor with the device by producing a specification for a finite-state machine to generate and/or sense any events that cannot be handled directly by the software routines. Note that one such finite state machine is shared among all implemented device routines. The synthesis of the hardware FSM is not covered, as several sequential logic synthesis tools are available for this task.

Our combined partitioning and scheduling problem is different from other scheduling problems because we schedule a set of signal events over two different "processors" with different cost metrics – the microprocessor itself and the additional FSM hardware. In software, the costly resource is time, which is represented by the individual events; in hardware, cost is dominated by area, which is closely related to the number of distinct signals on which events occur.

We make the following simplifying assumptions. First, we assume that the microprocessor can issue port read and write instructions with constant, regular spacing in time. Second, we assume that all driver sequences are executed atomically by the processor, possibly in response to an interrupt from the device, and cannot be overlapped or otherwise interrupted. These are both acceptable assumptions in the domain of real-time embedded controllers.

We can now provide a more formal description of a simplified version of the combined partitioning/scheduling problem. Given

- a set of signal wires, the inputs and outputs of the device,
- a set of events, with each event assigned to occur on particular wire(s),
- timing constraints specifying maximum and minimum separations between the events, as obtained from the device's databook, and
- a *processor scheduling quantum*, the time separation between successive instructions on the microprocessor,

compute a valid *schedule*, if one exists, such that all driver events are implemented in either interface hardware or software, and such that all scheduled times of events meet the given timing constraints. Each event implemented in software is assigned a unique time slot which is a multiple of the processor scheduling quantum.

5.2 A Flow-Based Approach to Partitioning and Scheduling

Creating such a partition begins by first determining if the driver events can be completely scheduled in software. If so, then there is no need to partition the events between hardware and software. However, when we must partition the events we will rely upon a Kernighan-Lin-style [9] iterative improvement algorithm on top of a max-flow/min-cut-based partitioning technique in the spirit of Bui *et al* [10]. Details of this algorithm are provided in Walkup and Borriello [11].

The input to the min-cut algorithm consists of a graph with weighted edges and two distinguished nodes, the *source* and the *sink*. The min-cut of a graph divides graph nodes into two partitions such that the source and sink are in different partitions and such that the total weight of the edges crossing the partitions is minimized. This is particularly appropriate since we already have a natural choice of the source and sink for our set of events – events which occur in hardware, such as data provided in a read operation, and events which must originate in software, such as data writes. Nodes in our min-cut graph represent the events to be partitioned and scheduled, and the weights of edges between them are heuristically determined to encourage a schedulable partition while generating as small an interface FSM as possible.

We begin by scheduling all events on the microprocessor, but initially allow more than one event to be scheduled in a single time slot. To encourage driver events to be scheduled in software whenever possible, we add edges connecting all driver events to the software sink. In areas of "high congestion" (i.e. where there are more events than instruction slots), we encourage one or more events to be pulled into the hardware partition by adding edges between them and the hardware "source". Furthermore, we create edges between events occurring on the same signal wire so that when events are moved to ease congestion, those whose signals are already in hardware will be more likely to move to hardware. We do this to minimize the size of the interface FSM, since its size grows with the number of signal wires it reads and/or writes. Events sharing narrow timing constraints have edges between them with weights inversely proportional to the constraint; these edges, in conjunction with the "high congestion" edge weight, encourage tightly timed events to be moved to hardware. The min-cut algorithm is then applied to this graph, and those events remaining in software are re-scheduled. This continues until a feasible schedule is found.

Figure 8 gives a graphical representation of the events and capacity relationships for the timing diagram of Figure 6. In this example, if the driver events cannot be feasibly scheduled on the microprocessor and some event must be moved to hardware, an appropriate choice would be moving the two reads of the Ready signal because they are less connected to other driver events.

Although it is possible to generate a separate hardware finite state machine for each device call used, greater hardware minimization is possible if we combine all finite state machines for such a device into one large FSM. This means that what might be an optimal software-hardware partitioning of driver events for a single device call may not be optimal for the entire device. More specifically, while it is *events* which we must partition between hardware and software, if we wish to minimize interface hardware size, we must minimize the number of distinct signals that appear in the interface logic. Therefore, one would apply this technique to all device routines for a single device at once.

Once the partitioning has been determined, we must generate the software routines and possibly a hardware state machine. Both of these tasks are straightforward. The software is essentially scheduled at the end of the iterative improvements. The hardware state machine can be constructed directly from the specification of the events on the signals to be generated by the hardware. More sophisticated FSM synthesis methods will be required when complex and tight timing constraints are involved.

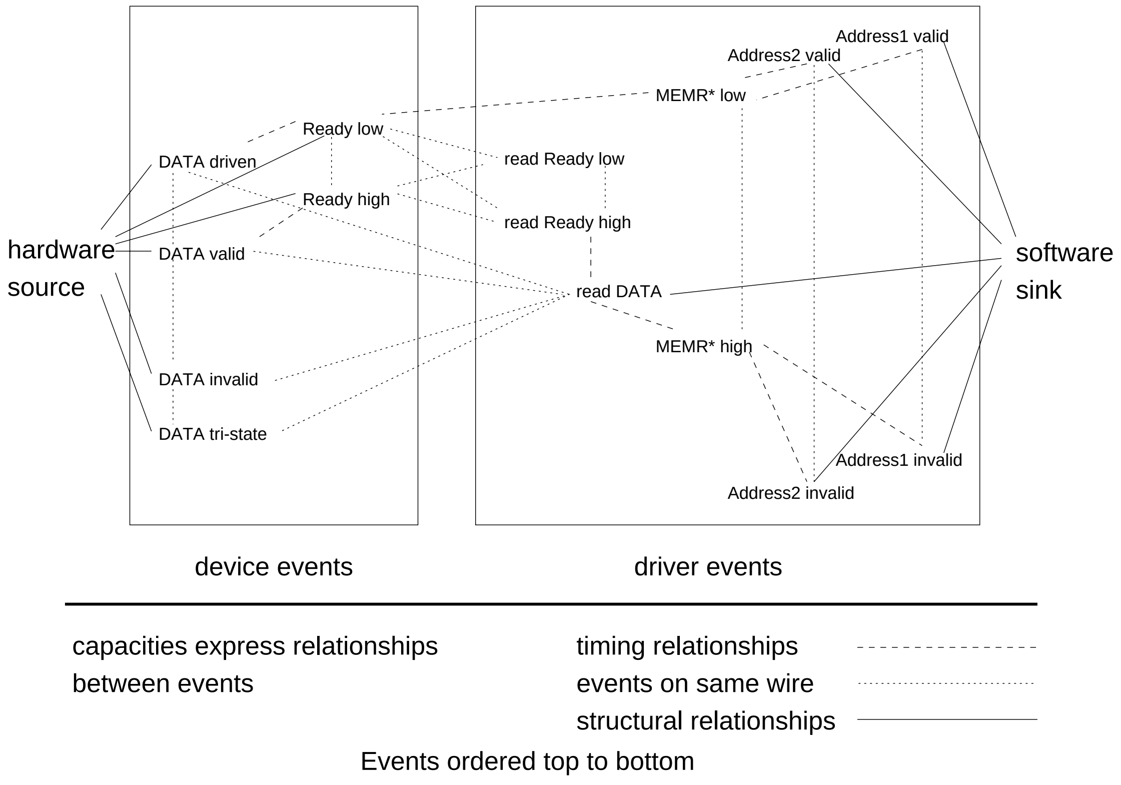


Figure 8: Flow solution for ISA bus read function.

6 Closing Remarks

We have outlined some issues in the scheduling of code for reactive real-time systems. Timing constraints are divided into two principal categories: high-level constraints arising from functional specifications that include rate, sequencing, and response requirements; and low-level constraints arising from the signaling protocols of the peripheral devices. This separation is a natural one as constraints within each of these two domains generally differ by orders of magnitude. Furthermore, tight low-level timing may require the addition of interface logic to handle some interactions for the processor. These differences lead to different methods being applied to solve the problems at the two levels.

The ideas presented here have led to algorithms that are being implemented in the Chinook hardware-software co-synthesis system under development at the University of Washington. The initial goal of the project is an automatic mapping tool that takes a high-level specification of an embedded system's functionality and its performance requirements and maps them onto a specified collection of processors and peripheral devices, which are described in detail in a library. The output provides all the elements needed to construct and program the system to operate as intended. Currently, the first complete version of Chinook is operational and can handle designs with one processor and user-specified partitioning. Thus, we have initial versions of all the pieces described in Figure 2 except for automatic partitioning.

The next steps include introducing automated partitioning and taking a closer look at the synthesis of interrupt-driven code. In addition, we are applying Chinook to an ever larger array of embedded systems applications for validation purposes as well as to help identify new directions for research.

References

[1] Richard Gerber and Seongsoo Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings of the Real Time Systems Symposium*, December 1993.

[2] R. De Simone F. Boussinot. The Esterel language. *Proceedings of the IEEE*, 79(9), September 1991.

[3] D. Harel. StateCharts: a visual formalism for complex systems. *Science of Programming*, 8, 1987.

[4] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 1991.

[5] Pai Chou, Ross Ortega, and Gaetano Borriello. Synthesis of hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1992.

[6] Pai Chou and Gaetano Borriello. Software scheduling in the co-synthesis of reactive real-time systems. In *Proceedings of the Design Automation Conference*, June 1994.

[7] David C. Ku and Giovanni De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design*, 11(6), June 1992.

[8] Gaetano Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, University of California, May 1988. Report No. UCB/CSD 88/430.

[9] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, February 1970.

[10] T.N. Bui, S. Chaudhuri, F.T. Leighton, and M. Sipser. Graph bisection algoritms with good average case behavior. *Combinatorica*, 7(2), 1987.

[11] Elizabeth A. Walkup and Gaetano Borriello. Automatic synthesis of device drivers for embedded systems. Technical report, University of Washington, April 1994.

[12] Edward Solari. *ISA and ESA, Theory and Operation.* Annabooks, 1992.

[13] Bruce Gladstone. Specification of timing in a digital system. *ASIC and EDA*, pages 46–52, August 1993.

Acknowledgements

This work was supported by an NSF Graduate Fellowship (Walkup), a PYI Award (MIP-8858782), and by the DARPA/CSTO Microsystems Program under an ONR monitored contract (N00014-91-J-4041).