Automatic Synthesis of Device Drivers for Hardware/Software Co-design

Elizabeth Walkup, Gaetano Borriello

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 94–06–04
August 16, 1994

Automatic Synthesis of Device Drivers for Hardware/Software Codesign

Elizabeth A. Walkup[*], Gaetano Borriello[†]

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{walkup, gaetano}@cs.washington.edu

August 16, 1994

Abstract

Automatically synthesizing device drivers, the hardware and software needed to interface a device to a processor, is an important element of hardware/software co-design. Driver software consists of the sequences of instructions needed for the processor to control its interactions with the device. Driver hardware consists of the digital logic necessary to physically connect the devices and generate signal events while meeting timing constraints. We describe an approach that begins with device specifications in the form of timing and state diagrams and determines which signals can be controlled directly from software and which require indirect control through intervening hardware. Minimizing this hardware requires solving a simultaneous scheduling and partitioning problem whose goal is to limit the number of wires whose events are not directly generated by the processor software. We show that even the simplest version of this problem is NP-hard and discuss a heuristic solution that should work well in practical situations.

[*]Supported in part by an NSF Graduate Fellowship.

[†]Supported by PYI Award (MIP-8858782) and by the DARPA/CSTO Microsystems Program under an ONR monitored contract (N00014-91-J-4041).

1 Introduction

In synthesizing the hardware and software that make up an embedded control application, timing constraints on the system arise from two main sources: the low-level signaling requirements of the peripheral devices the microcontroller interacts with, and high-level performance constraints such as response time and execution rate constraints. Since the performance constraints are generally expressed on a time scale an order of magnitude larger than the signaling requirements, it is natural to separately handle the two sources of timing constraints. These low-level signaling constraints can be satisfied by creating *device drivers* for each peripheral device. Such drivers consist of a script of software instructions for the microcontroller, and interface hardware between the microcontroller and peripheral devices. These device drivers allow the higher-level software, which implements system functionality, to invoke the appropriate signal sequence to interact with a device correctly without explicitly attending to the hardware and timing details of the peripheral devices. This creates a useful layer of abstraction between hardware requirements and system performance requirements while permitting optimization within the device drivers themselves. The problem of determining the best implementation of the driver – so that it meets the 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 [Bor88] considered the problem of generating glue logic to interconnect devices whose interfaces were specified by timing diagrams. The synthesized logic was guaranteed to meet all devices' timing requirements. 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 several reasons including: 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.

Furthermore, different processors with different speeds, instruction sets, and I/O ports will change the possible and preferable content of the software routines. Though it is possible to write a standard suite of device driver routines for each device-processor combination, there are several reasons why it may be advantageous to synthesize new drivers for each new use of a device. For example, an application which uses only a subset of a device's functionality may not require as much interface hardware as one using a comprehensive set. In addition, 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 pushed to hardware.

A device interface specification consists of a collection of timing and state diagrams explaining the exact sequence of events required on the device wires and the timing relationships between those events. Different sequences of events are used to execute the various device operations (e.g., read or write). The designer must analyze these specifications and determine which wires can be controlled directly from software and which require external digital logic. The external logic may take the form of straightforward memory-mapped register or decoders and multiplexors to overcome the limited I/O ports of a microcontroller. Sequential logic in the form of a finite-state machine is required when the device interface specifies events that are too close together in time to be generated or sensed by the processor.

This report is composed of six sections the first of which was this introduction to the problem. The next section considers a simple example to help illustrate the main concepts in the paper. Section 3 provides a formal description of the combined partitioning/scheduling problem along with an illustration of why even a simplified version is NP-hard and section 4 discusses an approach to solving this problem. Section 5 describes the role of this tool in the Chinook hardware-software co-synthesis system under development at the University of Washington. Finally, section 6 concludes and discusses our further plans.

2 Device Driver Synthesis Issues

Device behavior is generally described with timing diagrams as shown in Figure 1 for the ISA bus ready read operation. In this example, a simple software device driver would read and write from its i/o ports directly connected to device inputs and outputs to cause the following steps:

- provide data for *Address1* and *Address2*
- drive *MEMR** low
- sense the request ack (poll *Ready* until its value was 0)

- sense the valid data ack (poll *Ready* until its value was 1)
- read word from *DATA* line
- drive *MEMR** high

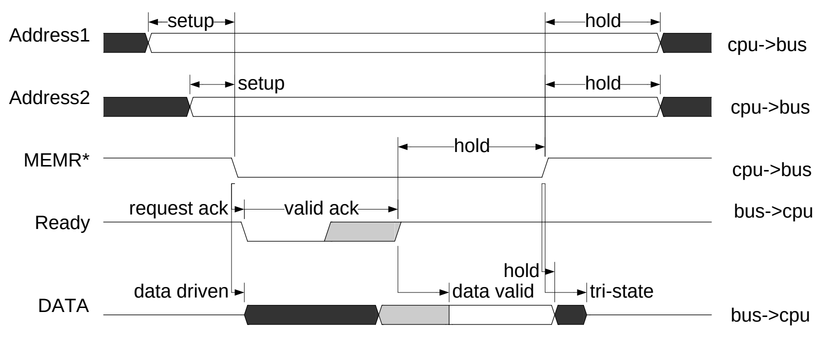
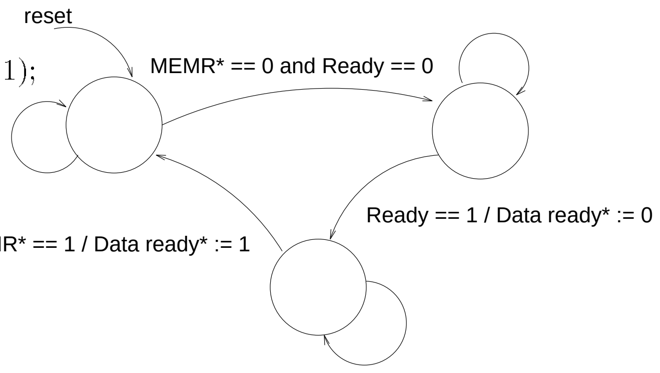


Figure 1: Timing diagram for ISA read operation.

Suppose, however, that the amount of time between the read acknowledge and valid acknowledge 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. Were this event missed, the microcontroller would have to assume that the read command it issued was not serviced by the bus. However, we could introduce a small finite state machine to watch the *MEMR** and *Ready* lines and lower a new signal *Data ready** once the valid ack has occurred. The microcontroller could then poll this new signal at its leisure to determine if the data is valid. Figure 2 shows a possible device driver for this scenario. It consists of: microcontroller code, a hardware finite state machine, port allocation information, and a depiction of the new timing relationships between signals in a timing diagram.

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. Creation of the device driver suite consists of three steps. First, for each device functionality 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 intermediate signals controlling the interactions between the software and external hardware. Second, we schedule the software events into subroutines corresponding to each operation of the device. Third, we

```
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

```
PortAllocation :
     Port1 == Address1
     Port2 == Address2
     Port3, bit 0 == MEMR*
     Port3, bit 1 == Data ready*
     Port4 == DATA
```

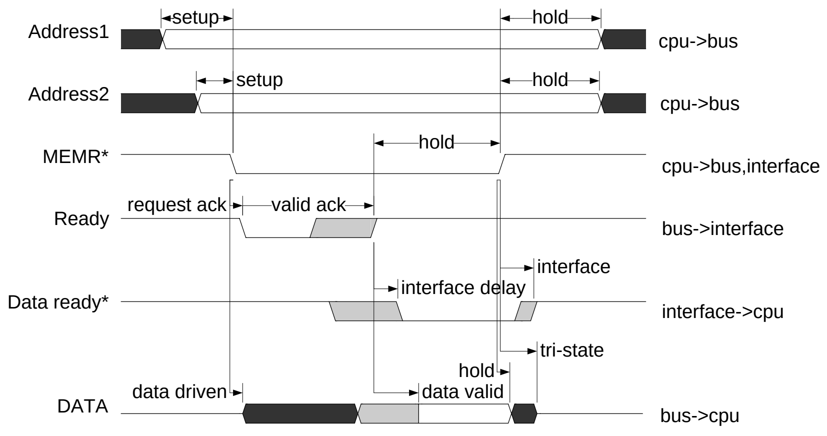


Figure 2: Device driver for ISA read operation, consisting of a microprocessor routine and hardware finite state machine.

generate a specification for a finite-state machine that will be used to interface the processor with the device and generate and sense any events that we have determined cannot be handled by software directly. Note that in actuality one such finite state machine is shared among all device functionalities used. Issues of port allocation or memory-mapping are handled by the algorithm described in [COB92] and are outside the scope of this paper. The synthesis of the hardware corresponding to the FSM is also not covered; it is assumed that sequential logic synthesis tools are available for this task.

3 Formal Problem Definition and NP-completeness Reduction

In effect, we have the problem of scheduling a set of signal events over two different "processors" – the microprocessor itself and the additional hardware we generate. This problem is distinguished from other scheduling problems as we are partitioning events into two radically different domains: software, in which the costly resource is time, represented by the individual events, and hardware in which it is area which is closely correlated to the number of signals to be generated or sensed.

In our treatment, we make a few 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 called atomically by the processor (possibly in response to an interrupt from the device) and cannot be overlapped or otherwise interrupted. These are all 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, $W = \{w_1, w_2, \ldots, w_k\}$, the inputs and outputs of the device,
  For the timing diagram of Figure 1 these signals are Address1, Address2, MEMR, Ready, and DATA.*

- a set of events, $E = \{e_1, e_2, \ldots, e_n\}$, with each event assigned to occur on one wire or bundle of wires,
  For signal Ready, the events generated by the device itself are the down and up transitions of the signal indicating request acknowledge and data validity. In addition, there are two implied events required of the

driver, namely reading a zero value on the Ready line, and then reading a one value.

- Δ, a relation of integral maximum skews between events in E such that $\Delta(e_i, e_j) = \delta$ implies that e_j may be scheduled no later than δ time units after e_i, that is, $t(e_j) \leq t(e_i) + \delta$ (minimum separations are represented with negative δ values); such constraints and guarantees specifying maximum and minimum separations between the events as obtained from the device's data book, and
- a *processor scheduling quantum*, q, the time separation between successive instructions on the microprocessor,

determine whether a *schedule* exists for the events such that

- driver events are partitioned between interface hardware and software;
- those events that are scheduled in software are assigned times which are multiples of q, the processor scheduling quantum, with at most one event per such time slot,
- and all scheduled times of events meet the given timing constraints.

In this partitioning, we further attempt to minimize the number of distinct signal wires on which some event assigned to the finite state machine occurs in order to minimize the amount of interface hardware generated.

If we simplify this problem to one in which we consider only devices with static timing behavior (i.e., the processor never needs to sense device outputs in order to determine when to schedule another event) we can show that it is still NP-hard. As would be expected, the scheduling portion alone of this problem is NP-complete. (The transformation is from 3-Partition as in [GJ79, GJ77] with timing constraints Δ used to build jobs of appropriate duration.) Appendix A gives a reduction from 3-SAT in which the event scheduling subproblems are extremely simple. This suggests that the partitioning portion of the problem is also difficult.

4 A Flow-Based Approach to Partitioning and Scheduling

In designing an approach to solving the partitioning portion of the combined partitioning and scheduling algorithm, two methods most readily come to mind: simulated annealing [KGV83], and the Kernighan-Lin [KL70] algorithm. However, neither of these methods are particularly appropriate. Neither simulated annealing nor Kernighan-Lin perform well on graphs with small degree [BCLS87], as is the case with the constraint graphs induced by the timing diagrams. Furthermore, if we are to combine the partition and scheduling steps, we need a method which can accommodate changes in the cost of moving events to hardware as will occur when events cannot be scheduled on the microcontroller. The Kernighan-Lin method uses a cost measure which is straightforwardly updated between iterations. In addition, given a good initial partition, Kernighan-Lin does perform well [BHJL89]. Thus we hope to solve the partitioning portion of the problem using a Kernighan-Lin [KL70] style iterative improvement algorithm on top of a flow-based initial partition inspired by [BCLS87].

The graph bisection method of [BCLS87] for graphs with small ($o(\sqrt{n})$) bisection width defines a *neighborhood* $N(v)$ of a vertex v to be all nodes within a given distance ($\log_{d-1}(\sqrt{n}-2)$ where d is the degree of the nodes in the graph) of v. It then finds the mincut for all pairs u, v where u and v's neighborhoods in the graph are respectively replaced with an infinite capacity source and sink. If the minimum such cut is indeed a bisection it is output as such. For graphs with small regular degree d, they show that this method nearly always works.

Our problem differs from this in several respects, but it is our intuition that these differences will not be great liabilities. We wish to find a partition of the graph such that one of the pieces is schedulable on the microcontroller, and the other is of smallest size possible given this condition, and thus we will not require that the routine ever return an exact bisection. We know that certain events in our graph naturally belong to one half of the partition or the other. For example, device-generated events always occur in the hardware portion of the system, and reading device-generated data most naturally occurs on the microcontroller, since to do otherwise requires significant additional hardware to store the value until the microcontroller later reads it. This indicates that our u and v for the bisection algorithm are already determined – one simply collapses all events that naturally occur in hardware into a source with infinite supply, and all events that naturally occur on the microcontroller are collapsed into a sink with infinite demand. Although our graphs are not d-regular, they will generally be of small degree.

In forming our partitioning problem edges in the flow graph can arise in three ways:

- All events on the same wire will have connecting edges to each other to encourage them to be placed in the same portion.

- At first, all events will be connected to the microcontroller source to encourage the events to remain scheduled on the microcontroller.

- Events with tight timing constraints can be encouraged to occur in the same portion by introducing edges with weight proportional to the inverse of the timing constraint.

- When scheduling events on the microcontroller, those in areas with "high congestion" (i.e. more events need instruction slots than are available) will have edges added between themselves and the hardware "sink", in hopes that one or more of them will be pulled into the hardware portion.

We would begin by performing the max-flow mincut algorithm followed by Kernighan-Lin iterative improvement using edges of the first three types mentioned above, and then attempt to schedule the microcontroller events using a scheduling method as in [KD92]. If the events cannot be scheduled without overlap, then edge weights and the Kernighan-Lin gain measures are updated as in the fourth condition above, and the iterative improvement continues.

Figure 3 gives a graphical representation of the events and capacity relationships for the timing diagram of Figure 1. Note that this figure shows that if lack of instruction slots on the microprocessor indicated moving some event to hardware, moving the two reads of the *Ready* signal from software to hardware would be most appropriate since they are less connected to other driver events.

Figure 4 shows what the flow diagram might look like for a device with three different operations that appear in the system specification.

- Each of the three rows represent one of the three driver calls.

- Nodes of different colors represent events occurring on different wires.

- Nodes in the leftmost boxes represent events generated by the device hardware.

- Nodes in the rightmost column of boxes represent events generated outside the device and which may therefore occur either in new interface hardware or software.

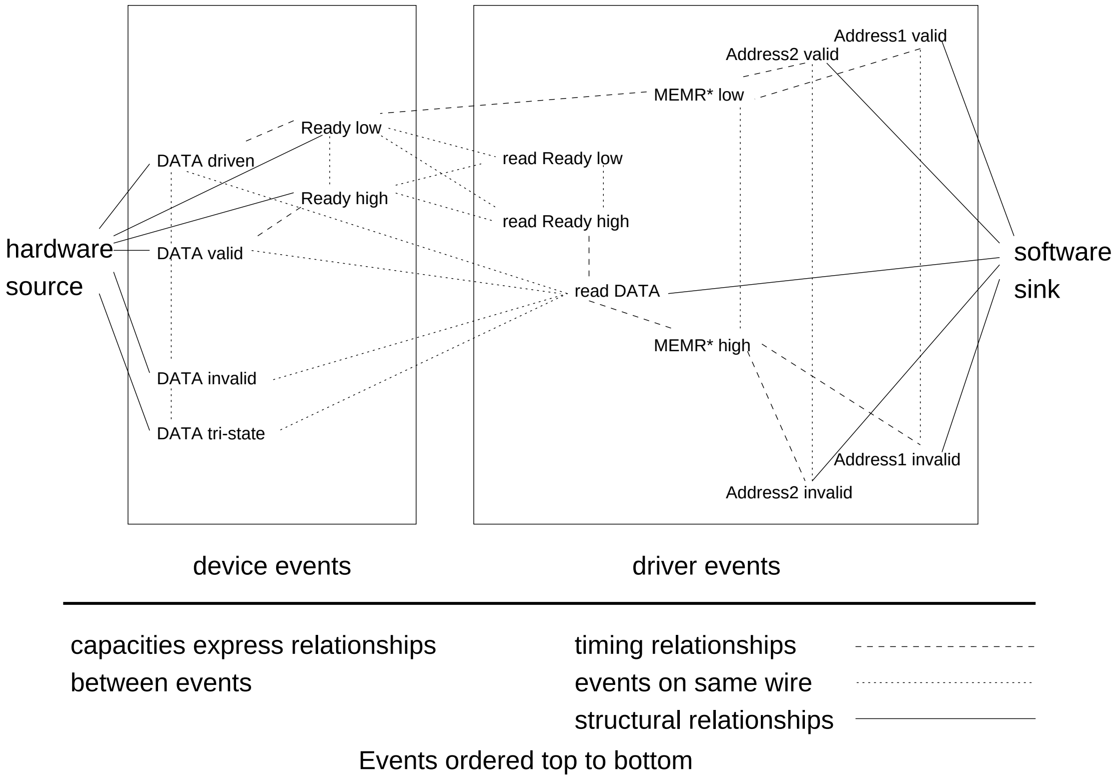


Figure 3: Flow solution for ISA bus read function.

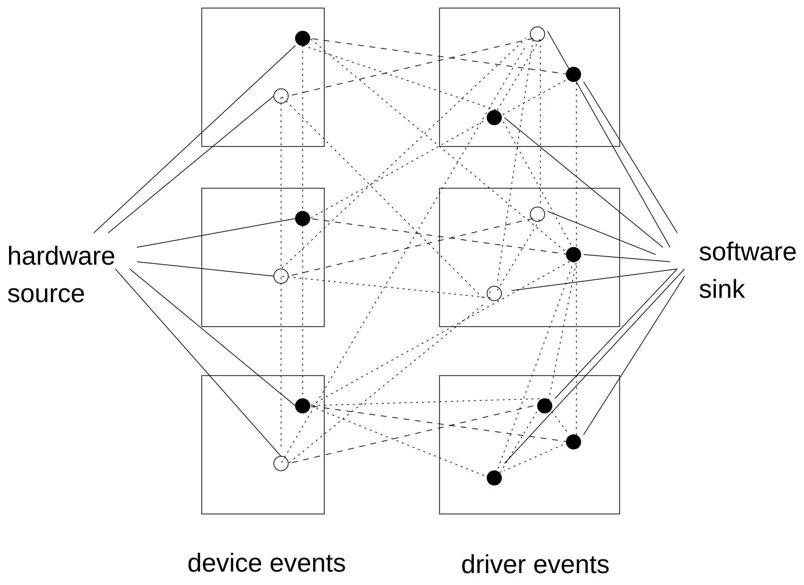


Figure 4: Flow solution for several device functionalities.

The depiction above shows the situation in which all events not generated by the device (i.e., the driver events in the rightmost column) can be generated directly from software at the appropriate times The cut has the device events on one side and the driver events on the other. Note that if there were not enough instruction slots to schedule in software all the events in the device call corresponding to the middle row then it may make more sense to move the "white" signal, and thus all its events, to hardware. This will cause fewer of the device's events to be generated by hardware, even though it moves more events from the given device call to hardware.

Once the partitioning has been determined we must generate the software routine 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.

5 Device Driver Synthesis in a Co-Synthesis System

Device driver synthesis is a critical task in hardware-software co-synthesis. It is executed as soon as a designer selects the processor and devices to be used in the design to be synthesized. The result, as we have seen, is a set of device driver routines and additional hardware needed for proper interfacing of the devices to the processor.

Device specifications are stored in a library in a form that is independent of any processor considerations. There is a formalized timing diagram [Bor92] for each device operation. This consists of a hybrid state and timing diagram annotated with timing constraints. In addition, each device also includes information as to which device inputs can share a microcontroller port without interfering with each other. This information is used by the port allocation step ([COB92]) to ensure that a device is not triggered to execute an operation inadvertently.

During device driver synthesis we must consider the effects that port allocation will have on the instruction sequences. It is often the case that the number of device wires that must be connected to the processor greatly outnumbers the number of available ports. However, if ports are judiciously shared then one microcontroller port may be connected to several device ports. To accomplish this, additional hardware to latch, tri-state, multiplex, or decode signals may be necessary. The effect of this is that it may no longer take just a simple instruction to alter or sense the value on a device wire. Instead, the device driver routine must now orchestrate the external hardware to have the desired effect. This may now require multiple instructions where one may have sufficed.

In our device driver synthesis approach, the device drivers are written first without assigning microcontroller ports to device ports. A separate port assignment routine [COB92] makes the assignments with the help of the sharability information. After this step is completed, the device driver routines can be further refined to reflect the port assignment. To accommodate the extra instructions that may be required, the sequence of signaling instructions includes information about where the sequence can be *stretched* to accommodate the extra steps.

Certain port allocations may be discarded if the device driver routine cannot be stretched in the necessary places. This can be viewed as an additional scheduling step that could not be handled prior to knowing the details of the port allocation. If no port allocation is feasible due to an overly tight driver routine, it may be necessary to go back to the synthesis procedure and modify some of the timing constraints. Re-synthesis may now lead to a different partitioning and scheduling that may move more signals to hardware in order to be able to accommodate port allocation.

6 Conclusion and Plans

We have presented the problem of automatically synthesizing device driver routines as a step in the design of embedded controllers. A possible solution was sketched that exploits algorithms from graph theory to partition and schedule interface events. Our plans are to implement these ideas in the Chinook Hardware/Software Co-synthesis System under development at the University of Washington. Our first step is to develop a device library format that represents the timing diagrams for the devices. We will then interface the results of device driver synthesis with the port allocation step including the feedback to generate a different partition or schedule when a port allocation cannot be accommodated. A description of the complete system is available in [CWB94].

7 Acknowledgements

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

References

[BCLS87] T.N. Bui, S. Chaudhuri, F.T. Leighton, and M. Sipser. Graph bisection algoritms with good average case behavior. *Combinatorica*, 7(2), 1987.

[BHJL89] Thang Bui, Christopher Heigham, Curt Jones, and Tom Leighton. Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

[Bor88] 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.

[Bor92] Gaetano Borriello. Formalized timing diagrams. In *Proceedings of the European Design Automation Conference*, March 1992.

[COB92] 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.

[CWB94] Pai Chou, Elizabeth A. Walkup, and Gaetano Borriello. Scheduling for reactive real-time systems. *IEEE Micro*, July 1994.

[GJ77] Michael R. Garey and David S. Johnson. Two-processor scheduling with start times and deadlines. *SIAM Journal on Computing*, 1977.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[Gla93] Bruce Gladstone. Specification of timing in a digital system. *ASIC and EDA*, pages 46–52, August 1993.

[KD92] 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.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, pages 671–679, May 1983.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, February 1970.

[Sol92] Edward Solari. *ISA and ESA, Theory and Operation*. Annabooks, 1992.

A NP-Completeness Reduction

The following reduction of 3-SAT to our combined partitioning and scheduling problem, has event scheduling subproblems which are extremely simple. This suggests that the partitioning portion of the problem is also difficult. We define 3-SAT with a set of Boolean variables, $U = \{u_1, u_2, \ldots, u_n\}$, and a set of disjunctive clauses $C = \{c_1, c_2, \ldots, c_j\}$, with individual clauses $c_j = (y_{j_1} + y_{j_2} + y_{j_3})$ where $y_{j_*} = u_i$ or $y_{j_*} = \overline{u_i}$ for some $u_i \in U$. The reduction then works as follows:

- there is a single "reference" wire and event, w_0 and e_0 with e_0 assigned to occur on wire w_0;
- for each variable x_i, there are two wires, w_i and $\overline{w_i}$;
- for each variable x_i there are two events ϵ_i and $\overline{\epsilon_i}$ with ϵ_i assigned to w_i and $\overline{\epsilon_i}$ assigned to $\overline{w_i}$;
- for i from 1 to n
  $\Delta(e_0, \epsilon_i) = \Delta(e_0, \overline{\epsilon_i}) = i$ and $\Delta(\epsilon_i, e_0) = \Delta(\overline{\epsilon_i}, e_0) = -i$;
- for each clause c_j there are three events, e_{j_1}, e_{j_2}, and e_{j_3} with e_{j_k} assigned to wire w_i if $y_{j_k} = u_i$ and e_{j_k} assigned to wire $\overline{w_i}$ if $y_{j_k} = \overline{u_i}$;
- for $i = 1$ to n and for k from 1 to 3
  $\Delta(e_0, e_{j_k}) = 2j + n$ and $\Delta(e_{j_k}, e_0) = -(2j + n - 1)$;
- $q = 1$;

If our combination scheduling and partitioning algorithm can solve this problem with exactly n wires moved to hardware, then there exists a satisfying solution to the formula. To see this, note that the first four items above indicate that at least one of $\{w_i, \overline{w_i}\}$ must be moved to hardware. Moving a wire w_i ($\overline{w_i}$) to hardware is equivalent to literal w_i ($\overline{w_i}$) being true. The last four items above ensure that at least one literal from each clause will be true by forcing an event on the corresponding wire to be moved to hardware.

This is done by forcing the scheduling of three events (one on each literal's wire) within a time period where only two slots are available.

It should be obvious that to verify a device driver routine's partition and schedule, one reforms the problem into that of "Is there a solution with only k wires moved to hardware?" and then verifies that all events in the schedule meet the requirements of Δ. Note that if the values in Δ are *consistent* (that is, there exists no set $E' \subseteq E$ with $E' = \{e'_0, e'_2, \ldots, e'_{k-1}\}$ such that $\sum_{i=0}^{k-1} \Delta(i, (i+1) \bmod k) < 0$) Then there always exists a solution to this problem by moving all events to hardware.