Run-Time Scheduler Synthesis For Hardware-Software Systems and Application to Robot Control Design

Vincent Mooney, Toshiyuki Sakamoto & Giovanni De Micheli
Computer Systems Lab, Stanford Univ., Stanford, CA 94305

Abstract

We present a tool that automatically generates a run-time scheduler for a target architecture from a heterogeneous system-level specification in both Verilog HDL and C. Part of the run-time scheduler is implemented in hardware, which allows the scheduler to be predictable in being able to meet hard real-time constraints, while part is implemented in software, thus supporting features typical of software schedulers.

We describe the tool flow and target architecture, synthesis of the control portion of the run-time scheduler in hardware, and control of the software using interrupts. Finally, we conclude with a sample application of the tool to a robot design example.

1 Introduction

Approaches to *hardware/software* co-design of embedded systems [2] can be differentiated in several ways. One way is to consider the system-level specification, which is either *homogeneous* (i.e., in a single specification language) or *heterogeneous* (i.e., involving multiple modeling paradigms). Another way is to distinguish how the CAD tool partitions the system specification: approaches consider either *fine-grained* partitions, i.e. at the operation or basic block level, or *coarse-grained* partitions, i.e. at the process or task level ([4] defines granularity in a slightly different way). For example, [3, 5] can be classified as *homogeneous* and *fine-grained* approaches, while [6, 7] are *heterogeneous* and *coarse-grained*, which is the approach we take in this paper.

There has been much previous work in partitioning [5, 7, 2]. However, system designs modeled by heterogeneous specifications are often already partitioned into modules or tasks. Whereas some optimality is lost in using a coarse granularity in partitioning, the resulting implementation is often closer to what designers expect, and interfacing hardware to software blocks is easier. We assume the availability of automated interface generation similar to [8, 7].

The sequence of hardware and software tasks can change dynamically in complex real-time systems, since such systems often have to operate under many different conditions. For example, a robotics system which comes into contact with a hard surface may have to change its force control algorithm, along with its attendant sensor set, estimators, and trajectory control routines. Therefore the scheduler must be dynamic.

In hardware-software codesign an important problem is the management of software routines and their coordination with hardware. A clear and easy solution is to put the run-time system in software and suitably design the hardware such that it can be controlled from the software. Unfortunately software schedulers may not be predictable as far as being able to satisfy real-time constraints. Therefore we implement the time-constrained portion of the scheduler in hardware, where delays are accurately known. This paper presents a strategy for a mixed implementation of dynamic real-time schedulers in hardware and software.

2 Motivation and CAD Requirements

We aim at supporting system-level design with hardware and software tasks custom written for a target architecture. We assume that system requires both static scheduling, especially in the coordination of the hardware components to meet real-time constraints, and dynamic scheduling (given the inexact delay of software). A *run-time scheduler* must meet both of these scheduling requirements. Our tool, called SERRA, automates the generation of the run-time scheduler, thus providing for the *synchronization* and *scheduling* of system-level components in hardware and software.

Our approach assumes a *coarse-grained* partition of the system to be implemented into medium sized components called *tasks*. A typical size of a task is around 50 to 200 lines of Verilog or C; however, the only limit on task size depends on the high-level synthesis tool or compiler chosen. SERRA automates the generation of a run-time scheduler to manage concurrent tasks executing in hardware or software. We refer to the former as *hardware-tasks* and the latter as *software-tasks*. Thus, SERRA provides the user with the ability to evaluate the performance of different partitions with an automatically generated run-time scheduler (system).

As a motivational example, consider a set of control laws used to calculate appropriate torques for a robot arm.

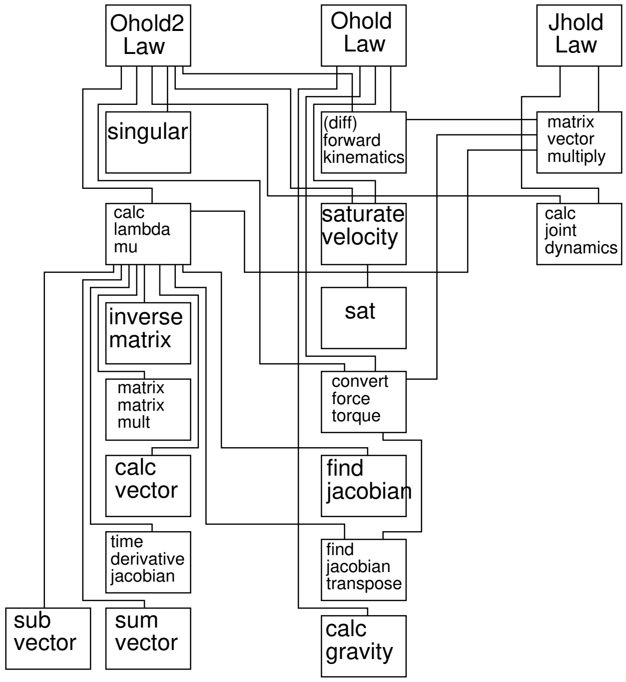

Figure 1: Robotics Example: Concurrent Control Laws

We assume that the controller manages two arms at the same time, and thus any two of the laws may be selected in each execution. An execution of the arm controller must complete once every millisecond. Figure 1 shows three of the ten different laws used with a PUMA arm; `Ohold2 Law`, `Ohold Law`, and `Jhold Law` are top-level tasks which call subtasks in a particular sequence. Some of the subtasks involve hardware components with timing constraints specified on a cycle basis.

The CAD requirements for co-design of this system are as follows. First, we need to satisfy hard real-time constraints imposed by some of the hardware components in the system as well as by external hardware. Second, we need to optimize the run-time system over calls to multiple tasks in hardware and software. This involves allocation of tasks to hardware and software as well as interface generation for communication. We want as much concurrency as possible in the final implementation. In this paper, we focus on the synthesis of a run-time scheduler.

3 Tool Flow and Target Architecture

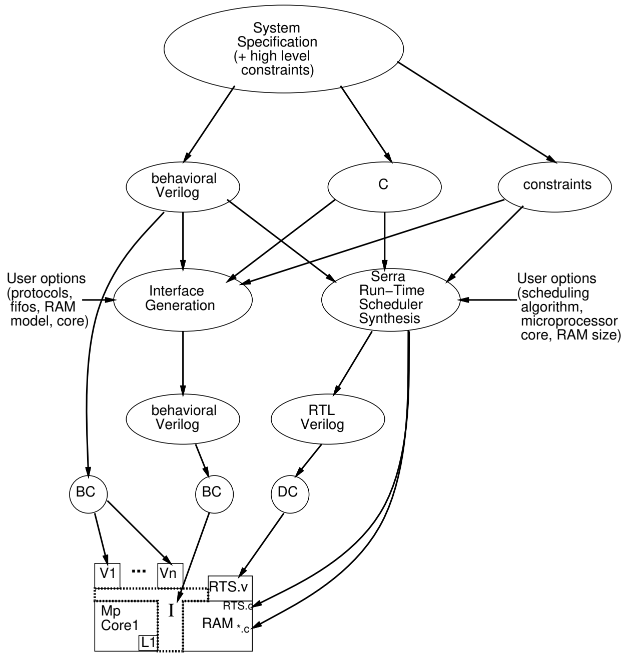

Figure 2: Tool Flow and Target Architecture

Hardware-tasks are specified in Verilog that can be synthesized by the Synopsys Behavioral Compiler^{TM}[1]; its use is labeled `BC` in Figure 2 (`DC` labels the Design Compiler^{TM}). Software-tasks are written in C. Microprocessor cores, memories (DRAM, SRAM), FIFO models, and other custom blocks are assumed as available inputs to the system.

The system-level tasks in Verilog and C, as well as constraints, are input to a tool that generates the interface and to SERRA. Constraints include relative timing constraints (minimum and maximum separation), resource constraints, and rate constraints. The overall control flow of the run-time scheduler is synthesized into hardware, while the necessary code for calling tasks in software is generated as well. Further aspects of a RTOS can be added in software by the user if desired.

4 Task Execution

We associate a *start* and a *done* event with each task in order to allow the scheduler to control the task. In hardware the two events are simply signals on an input port and an output port, respectively. For software, we have a *start* vector and a *done* vector which encapsulate the *start* and *done* events for each software-task.

Note that some tasks are called multiple times by different tasks, such as `matrix vector multiply` in our robot example, as can be seen in Figure 1. Some real-time constraints in hardware can be satisfied by high-level synthesis. However, constraints at the task level must be handled by the run-time system. How can the run-time system dynamically allocate tasks while at the same time predictably satisfying exact timing constraints between tasks?

We solve this scheduling problem using the algebra of control-flow expressions (CFEs) [9], which represent the serial/parallel flow of computation, branching, iteration, synchronization and exceptions. CFEs can specify control flow that satisfies our real-time constraints in hardware while also controlling dynamically the flow of execution. CFEs have a deterministic finite-state machine (FSM) semantics, and so can be compiled into specification FSMs representing the possible control-flow implementations.

The Behavioral Compiler^{TM} already performs detailed control synthesis for individual hardware-tasks. We thus apply CFEs at a higher level of abstraction: the coordination of tasks, with a single CFE action serving as the *start* event for a task in hardware or software. This contrasts with earlier uses of CFEs to model systems at the operation level[10]. Using CFEs to coordinate tasks hides the coordination of low-level operations from the CFE model and results in greatly reduced control logic.

4.1 Task Control Flow Extraction

SERRA takes as input a collection of tasks described in Verilog and C. We obtain a Control-Data Flow Graph for each task, or task-CDFG, from the input. Each node in a task-CDFG corresponds to a call to another task. The user must indicate which task is the root and thus kicks off execution and calls of the other tasks. An example of such a root task after obtaining its task-CDFG is seen in Figure 3, which executes appropriate control laws of Figure 1 to output torques for two PUMA robot arms. If a task does not call any other tasks, then its corresponding task-CDFG is empty.

The CFEs of the task-CDFGs capture real-time constraints between tasks (usually hardware-tasks) as well as mutual exclusion between dynamically called tasks. Currently the user must enter the constraints directly in CFEs. Note that the CFEs are extracted from Verilog or C. Clearly, the resulting CFEs correspond exactly to the coarse-grained partition of the system. Currently this CFE-extraction is performed automatically for Verilog and manually for C.

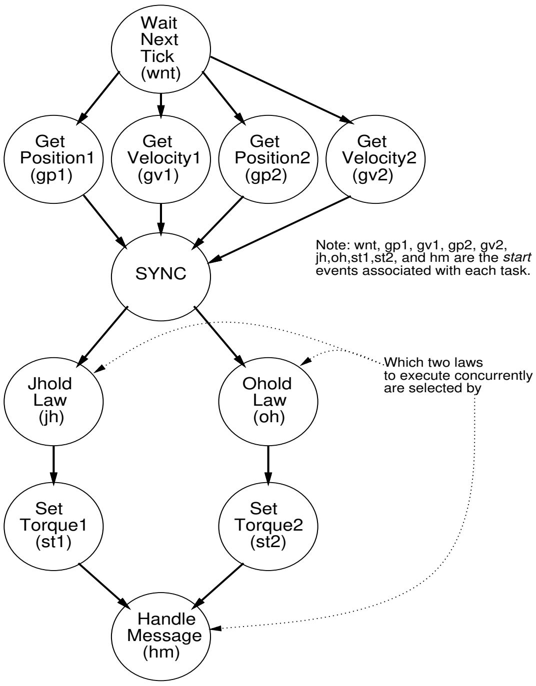

Figure 3: Robotics Example: Root-Level Task

Example 1[CFE Extraction] For our robotics example, we begin with the task specified by the user as root-level. In this case it is a routine written in C. After control- and data-flow analysis, we end up with the task-CDFG of Figure 3, where the $SYNC$ node is not a task but indicates that the previous tasks must complete before continuing. With each task that has nondeterministic delay, we associate a CFE control signal and a CFE action, e.g. Wait Next Tick has CFE control signal $c0$ and action wnt. Tasks with deterministic delays need only an action. The action represents the *start* event for that task in the synthesized system. Recall that in CFE semantics, * indicates zero or more cycles, ^{4} indicates exactly four cycles, $||$ indicates parallel execution, $\cdot$ indicates serial execution, and $^\omega$ indicates an infinite loop. The extracted CFE is
$((c0 : wnt)^* \cdot (gp1^4||gv1^4||gp2^4||gv2^4) \cdot (((c1 : jh)^* \cdot st1)||((c4 : oh)^* \cdot st2)) \cdot (c6 : hm)^*)^\omega$ □

4.2 Run-Time Scheduler Partition

From the consistent model (CFEs) of the task schedule we make a split of the run-time scheduler into hardware and software based on an analysis of the constraints. We hypothesize that exact relative constraints between tasks cannot be satisfied by software. Thus, we have the problem of choosing between the predictability of satisfying real-time constraints in hardware and the desirability of having some features of a RTOS. We try to accommodate both choices by putting in hardware a FSM corresponding to the CFE description of the system, while putting in software a reactive executive which calls the appropriate software-tasks when signaled by the hardware FSM.

Therefore we split the run-time scheduler into two parts:

- An executive manager in hardware with cycle-based semantics that can satisfy hard real-time constraints.
- A software routine scheduler that can execute different threads and may implement multitasking.

The flow of the Serra Run-Time Scheduler Synthesis tool is shown in Figure 4.

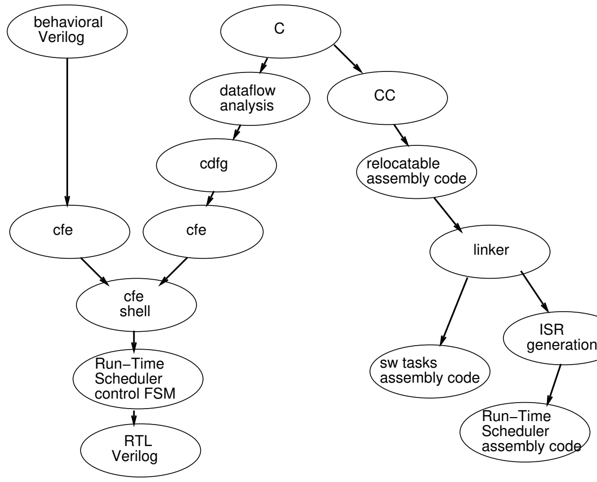

Figure 4: Block diagram of Serra

5 Serra Run-Time Scheduler Synthesis

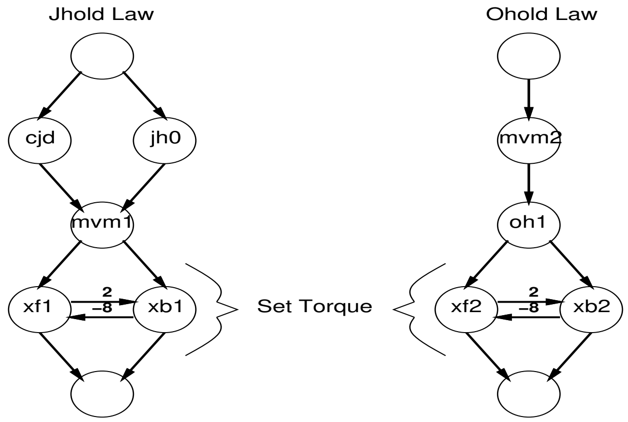

Figure 5: Task Control-Data Flow Graphs of Jhold Law, Set Torque, and Ohold Law

5.1 Control State Machine

CFEs are extracted from all the tasks input to Serra. Control inputs $c0, c1, c2, \ldots, cn$ are used to dynamically "request" resources. These control inputs are generated by a conjunction of *done* events of previous tasks in the control flow, except for $c0$ which can also be generated by a global RESET.

Example 2[CFE Extraction With Max/Min Timing Constraints] Consider the two tasks `Jhold Law` and `Ohold Law`. Their task-CDFGs, with `Set Torque` included (see Figure 3), are shown in Figure 5. Node cjd in corresponds to task `Calculate Joint Dynamics` and node $jh0$ corresponds to task `Jhold Law0`. `Set Torque` consists of two hardware modules, `Xmit Frame` (xf) and `Xmit Bit` (xb), which are used to input the torque data in a bitwise fashion. A minimum two-cycle delay is required between the start of xf and the start of xb, with a maximum allowable delay of eight cycles. Similarly, we assume a resource constraint in that only one `matrix vector multiply` (mvm) hardware unit is available.

The maximum and minimum delay constraints between xf and xb are easily specified in the CFE as $xf \cdot 0 \cdot 0^{<7} \cdot xb$ where 0 is a no-op and $^{<7}$ means that the operation can occur between zero and six times. Now the CFE extracted from the Verilog specification of Jhold Law is

$((c1 : cjd)^{*} || (c2 : jh0)^{*}) \cdot (c4 : 0)^{*} \cdot (c3 : mvm1)^{*} \cdot xf1 \cdot 0 \cdot 0^{<7} \cdot xb1$

And the CFE extracted from the C code for Ohold Law is

$(c4 : mvm2)^{*} \cdot (c5 : oh1)^{*} \cdot xf2 \cdot 0 \cdot 0^{<7} \cdot xb2$

Note that the control signal $c4$ for $mvm2$ is used to delay the execution of $mvm1$ in the case that $c3$ is asserted while $mvm2$ is still executing. This works provided $mvm2$ is always executed before $mvm1$. A more general way of providing for mutually exclusive dynamic granting of *start* events for the same resource is to use the CFE $NEVER$ sets[9]. □

The CFE of the root task guides the composition of the extracted CFEs. If the input tasks are a complete set for the system, then SERRA will successfully collapse all the extracted CFEs into the root CFE.

Example 3[CFE Composition] To implement the hardware portion of the run-time scheduler, the CFE of Example 2 and equivalent CFEs for the other control laws are embedded in the CFE of Example 1. The result is

$((c0 : wnt)^{*} \cdot (gp1^{4} || gv1^{4} || gp2^{4} || gv2^{4}) \cdot$
$(((c1 : cjd)^{*} || (c2 : jh0)^{*}) \cdot (c4 : 0)^{*} \cdot (c3 : mvm1)^{*} \cdot xf1 \cdot 0 \cdot 0^{<7} \cdot xb1)$
$||\ (\ (c4 : mvm2)^{*} \cdot (c5 : oh1)^{*} \cdot xf2 \cdot 0 \cdot 0^{<7} \cdot xb2\ ) \cdot (c6 : hm)^{*})^{\omega}$

Note that $(c1 : jh)^{*} \cdot st1$ and $(c4 : oh)^{*} \cdot st2$ in Example 1 have been replaced with the CFEs of Example 2

Control signals $c1 - c4$ in the CFEs are formed based on the control flow. For example, $c3$ is set high when the cjd_{done} and $jh0_{done}$ events have both occurred. (Note that $mvm1$ will not be activated, however, if $c4$ is high – see the comment at the end of Example 2.) With this composition, we obtain a CFE specification of the system which generates *start* events for each task (via CFE actions). □

SERRA synthesizes the control-unit of the scheduler by means of tool THALIA which takes as input a CFE description and produces a logic-level description in synthesizable Verilog[9, 10]. The constraints specified in the CFEs input to THALIA are translated into automata. Thus, for the control to be synthesizable, the intersection of constraints with possible state machine implementations must not be void.

5.2 Control of Software

Software has a *start* vector and *done* vector which implement *start* and *done* events for each software-task. If there are less than 32 distinct software-tasks, each vector can be contained in a single word with a simple one-hot encoding (otherwise more words can be used).

The hardware run-time scheduler updates the *start* vector in software as follows. First, it updates its local register containing the *start* vector. Then it triggers an interrupt on the CPU. The CPU interrupt service routine reads the register using a memory-mapped I/O read and places it into the software copy of the *start* vector.

When the software-task is finished executing, it updates the *done* vector by writing the value out with memory mapped I/O. Thus, the *done* vector in the run-time scheduler in hardware is updated. Notice that in the above two cases, a dedicated port could be used instead of memory-mapped I/O, depending on the CPU.

5.3 Software Generation

For the software that runs on the microprocessor core (CPU), the individual software-tasks are compiled and linked using standard C compilers and linkers. Memory-mapped I/O is called with C pointers set explicitly to the appropriate addresses. We thus end up with a set of software-tasks and their start addresses in the program code.

Therefore, given a particular value of the *start* vector, the appropriate software-task(s) can be executed. However, we need an executive for the software. We consider two options.

5.3.1 Option #1: Hardware Communicates *Start* Vector to Software

For this option, the software part of the run-time scheduler implements the following:

- A hardware interrupt updates the *start* vector.
- The hardware interrupt finishes.
- A polling executive in software reads the *start* vector in a $for(;;)$ loop.
- The executive picks a process to execute.
- When a software-task is finished, it writes out the new value of the *done* vector.

An advantage of this option is that it can support standard RTOS scheduling algorithms (round-robin, rate-monotonic, etc.). Multiprocessing is helpful when a long software-task executes at the same time as a short duration software-task, but a price is paid when switching context. A disadvantage is the slower response time due to added overhead for implementing the RTOS scheduling algorithm, polling executive, and associated context switches.

5.3.2 Option #2: Hardware-Driven Software Execution

The second option considered is as follows:

- Interrupt updates the *start* vector.
- Interrupt Service Routine (ISR) executes a jump to the appropriate software-task.
- When the software-task finishes, the *done* vector is written and the ISR finishes.

The main advantage of this option is faster execution. Some disadvantages are that there is no multiprocessing and all software-tasks are executed in kernel mode.

Background tasks, such as a user interface or output display, can run when the CPU returns from an interrupt.

6 Example and experimental results

For our example, we consider the robot control algorithm of Figure 3. We implement a subset of the tasks required for executing `Jhold Law` and `Set Torque` in parallel with `Ohold Law` and `Set Torque` (shown in Figure 5). Notice that `Xmit Frame` (xf) and `Xmit Bit` (xb) of `Set Torque` have a strict real time constraint that could not be satisfied with control signals generated by a run-time scheduler in software (note in Figure 2 we have an L1 cache). We assume that the full system drives `Xmit Bit` from hardware modules other than `Xmit Frame` and thus must be kept separate.

Since none of our software-tasks are exceptionally long, we choose Option #2. This provides the faster execution speeds, even though it serializes all software-tasks. The software tasks are compiled and linked into assembly, with data and program memory statically allocated, as well as memory-mapped I/O. Finally, the software portion of the run-time scheduler is generated in the form of an Interrupt Service Routine that reads in a *start* vector which task needs to be executed in software and then executes the task.

The system begins each iteration once a millisecond. First the run-time scheduler starts the execution of *mvm* in hardware for `Ohold Law`, *cjd* in hardware and $jh0$ in software for `Jhold Law`. Next it passes execution for `Ohold Law` onto software with $oh1$, and it passes execution for `Jhold Law` onto hardware with *mvm*. Finally, it serializes accesses to `Xmit Frame` and `Xmit Bit` to set the torques for the robot.

Software-Task	# Lines C	# Lines Assembly
jh0	17	151
oh1	178	695
run-time-sch-sw	23	41

Table 1: Code space for Jhold and Ohold Laws

Hardware-Task	# Lines Verilog	Area
cjd	598	14353
mvm	246	33965
xmit-frame	108	987
xmit-bit	66	199
run-time-sch-hw	391	314

Table 2: Results for the synthesis of hardware-tasks

Table 1 presents the results for the compilation of the software. In Table 2 we see the results for the synthesis of the hardware using the Behavioral Compiler^{TM}, except for the run-time scheduler hardware part which was synthesized with the Design Compiler^{TM}. Finally, the last column in Table 2 shows the number of gate equivalents the hardware required using the LSI 10K Logic library.

We simulated the run-time scheduler hardware unit in Verilog using Chronologic's VCS^{TM}. The resulting waveform with a particular test pattern can be seen in Figure 6. Signals *wnt*, $gp1, \ldots$, *hm* are the *start* events for the corresponding tasks in Figures 3 and 5. For the sake of simplicity, communication delays due to the interface are not included. With the interface delays included, the transitions between signals would be much longer.

Signals $c1$ - $c6$ are the dynamic requests for resources made to the scheduler. Note that although $c3$'s request for $mvm1$ comes at cycle 10 (at the dotted line in Figure 6), it is not granted until cycle 12 (i.e. $mvm1$ goes high at cycle 12) since before then $c4$ is high (see the comment at the end of Example 2). Further, note that sometimes, as in the request $c6$ for *hm* (`Handle Message` of Figure 3), the granting is delayed

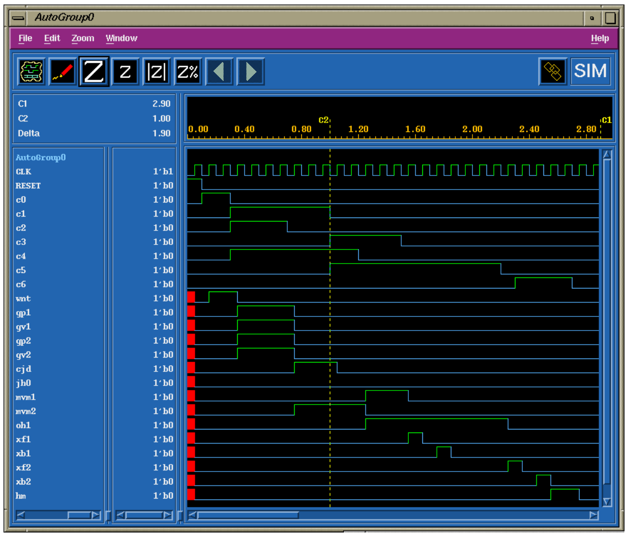

Figure 6: Waveform Display of Synthesized Scheduler FSM

automatically to meet the control-flow requirements. Thus, whenever there are requests for the same resource, the activations are serialized. Also note that `xf1` and `xb1` occur two cycles apart, the minimum specified.

7 Conclusion

The SERRA Run-Time Scheduler tool helps designers perform system-level design with hardware and software at a coarse level of granularity. We have shown how one can synthesize a run-time scheduler in hardware and software that can predictably meet real-time constraints while dynamically executing tasks in hardware and software. We have utilized the methodology of control-flow expressions to synthesize the hardware control portion of the scheduler.

For our future work we plan to address the issue of real-time analysis for the entire system as well as constraint analysis for subschedules of the system that involve both hardware and software.

Acknowledgements

ARPA sponsored this research under grant No. MIP DABT63-95-C-0049.

References

[1] D. Knapp, *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*, Prentice Hall, Upper Saddle River, NJ, 1996.

[2] G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, Kluwer Academic Publishers, Norwell, MA, 1996.

[3] J. Henkel, Th. Benner, R. Ernst, W. Ye, N. Serafimov and G. Glawe, "COSYMA: A Software-Oriented Approach to Hardware/Software Co-design," *The Journal of Computer and Software Engineering*, Vol. 2, No. 3, pp. 293-314, 1994.

[4] J. Henkel, R. Ernst, "The Interplay of Run-Time Estimation and Granularity in HW/SW Partitioning," *4th. Int'l Workshop on Hardware/Software Codesign*, Pittsburgh, 1996.

[5] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Boston, MA, 1995.

[6] Jay K. Adams and Donald E. Thomas, "Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems," *International Symposium on System Synthesis*, pp. 10-15, September 1995.

[7] D. Verkest, K. Van Rompaey, I. Bolsens & H. De Man, "CoWare–A Design Environment for Heterogeneous Hardware/Software Systems," *Design Automation for Embedded Systems*, Vol. 1, No. 4, pp. 357-386, October 1996.

[8] Pai H. Chou, Ross B. Ortega, and Gaetano Borriello, "The Chinook Hardware/Software Co-Synthesis System," *International Symposium on System Synthesis*, pp. 22-27, September 1995.

[9] C. N. Coelho Jr. and G. De Micheli, "Analysis and Synthesis of Concurrent Digital Circuits Using Control-Flow Expressions," *IEEE Transactions on CAD/ICAS*, Vol. 15, No. 8, August 1996 .

[10] V. Mooney, C. Coelho, T. Sakamoto and G. De Micheli, "Synthesis From Mixed Specifications," *European Design Automation Conference*, pp. 114-119, September 1996.