Path-Based Edge Activation for Dynamic Run-Time Scheduling

Vincent J. Mooney III
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
mooney@ece.gatech.edu

Abstract

We present a tool that performs real-time analysis and dynamic execution of software tasks in a mixed hardware-software system with a custom run-time scheduler. The tasks in hardware and software have control-flow constraints (precedence and alternative execution), resource constraints, relative timing constraints, and a rate constraint. The custom run-time scheduler dynamically executes tasks in different orders, based on the conditional execution path, such that a hard real-time rate constraint can be predictably met.

We describe the task modelling, run-time scheduler implementation, and real-time analysis. We introduce the concept of path-based edge activation utilizing conditional edges. We show how our approach fits into an overall tool flow and target architecture. Finally, we conclude with a sample application of the system to a design example.

1 Introduction and Motivation

For real-time embedded systems, designers must meet timing constraints in order for the design to be successful. System designers need tight bounds on execution delays in order to meet soft and hard real-time constraints. In hardware-software codesign, scheduling of coarse-grained resources (e.g., a hardware filter or a software routine implementing a filter) under timing constraints is a difficult problem due to the presence of parallel threads of execution in the application, with the same resource required by different threads. We consider in this paper the following formulation of the real-time analysis problem: we represent the system with a control/data-flow graph where the nodes represent software or hardware, the graph edges represent dependencies (precedence constraints), conditional paths exist in the graph (alternative execution), and the graph is invoked at a fixed rate (a rate constraint). We assume that to coordinate the system we use the *run-time scheduler* of [16, 18, 19].

Previous approaches to real time analysis have focused on software [2] since the performance analysis of ASICs is considered a well studied problem already. Rate Monotonic Analysis (RMA) [3] and Generalized Rate Monotonic Analysis (GRMA) [4] both assume that tasks are independent and that each task has its own period and deadline. RMA has been extended to account for release jitter and resource contention [5, 6]. RMA has also been extended to allow precedence among tasks by formulating the problem as a big task with the length of the least common multiple (LCM) of all the periods [7]. Unfortunately, this approach is usually impractical for hardware-software codesign[8].

Our formulation is similar to [1, 7, 8, 13]. However, in our case we synthesize a custom run-time scheduler in hardware and software for the application [16]. As a result, we have more information about the scheduling of hardware and software tasks. Given this more exact level of control, we can perform tight real-time analysis with very high CPU utilization. In performance- or safety-critical systems (e.g., a mobile robot control system) our approach can provide precise real-time bounds.

The scheduling problem addressed here is most similar to the scheduling problem addressed by Conditional Process Graphs[1]. In particular, the goal in [1] is to generate a static schedule which will minimize the execution time for any allowable value of the conditionals controlling alternative execution in the task graph. Since this may require activations of different tasks in different orders, they keep track of the possible paths using a *schedule table*. There may be a *conflict*, where, for example, the optimal schedule of one path requires that process P_3 be scheduled at time t_k, while the optimal schedule of another path requires that P_3 be scheduled at time t_l, $t_k \neq t_l$. Conflicts are handled by adjusting one of the path schedules, called *schedule merging*. In the approach in this paper, however, no such adjustment of the path schedule is necessary; thus, we may potentially produce a better (faster) schedule.

In our approach, if a solution is found, we add *conditional edges* to the system graph and guarantee that the system meets its relative timing constraints, control-flow (precedence and alternative execution) constraints, and

its rate constraint, assuming the system uses a custom run-time scheduler.

1.1 Terminology and Approach

We take an approach to system-level design where the designer or a partitioning tool has split the application into *coarse-grained* tasks which coordinate with each other. Each task consists of many lines of C or Verilog or another language – say, 50 lines or more. A task implemented in hardware is called a *hardware-task*, whereas a task implemented in software is a *software-task*.

We assume that some static scheduling is required, e.g., in the presence of relative timing constraints among hardware-tasks. We further assume that some dynamic scheduling is required to handle inexact execution delays, which are certainly present in software-tasks, and conditional execution of tasks. The run-time scheduler synthesis of [17, 18, 19] assumed that no conditional execution of tasks would occur.

Our tool, called CLARA2, automates the addition of *conditional edges* to the original control-flow of hardware- and software-tasks in order to satisfy resource constraints while minimizing *worst case execution time (WCET)* for the application.

The target designs for our approach are embedded systems. We assume the availability of mature high-level and logic synthesis tools for the generation of digital hardware as well as state-of-the-art compilers for the software. While our examples utilize tasks with 90 to 2000 lines of Verilog or C, the only limit on task size is determined by the synthesis tool or compiler chosen. Additionally, a task can be taken from an Intellectual Property (IP) library of pre-designed cores, in which case synthesis may not be necessary. We assume that tasks are customized for a target architecture. This approach matches design practice, where designers often describe their systems in a heterogeneous way, using description languages appropriate to the subsystem being implemented.

The rest of the paper is organized as follows. Section 2 talks about how we extract a graph of the system corresponding to control flow at the task level as opposed to the operation level. Section 3 presents the main contribution of this paper, *path-based edge activation*, which is implemented via *conditional edges*. Section 4 describes our implementation of the run-time scheduler. Section 5 presents the real-time analysis which generates *conditional edges* to add to the system CDFG. Section 6 describes the tool flow, target architecture and run-time scheduler synthesis. Section 7 gives some experimental results and presents an example from robotics. Finally, Section 8 concludes the paper.
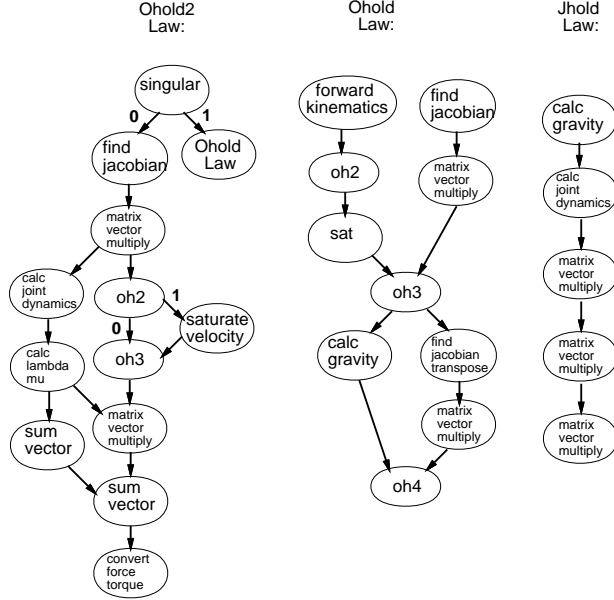

Figure 1: Robotics Example: Concurrent Control Laws

2 Task Modelling

The research here builds on previously reported results; thus, in this section, we briefly summarize the task modelling approach of [16, 17, 18, 19].

Tasks are specificied in Verilog or C, with one of the tasks designated as the *main task*. The main task begins execution and calls the other tasks. The main task specifies the overall sequence of tasks in the application (an example of a main task can be seen in Figure 2). From the main task we extract a Control/Data-Flow Graph (CDFG) representing the sequence of task invocations, where each action in the CDFG corresponds to a call to one of the tasks specified either in C or in Verilog. We assume that the CDFG corresponding to the main task does not contain any looping over tasks; only alternative execution (e.g., if/case) is supported in the main task.

We assume that we have a rate constraint specified for the CDFG representation of the system. In other words, we assume that the main task is invoked at a fixed rate.

Tasks executed on the same resource (for example, software-tasks executed on the same processor) are placed in a single $NEVER$ set. For example, $NEVER = \{x, y, z\}$ indicates that tasks x, y, and z can never be active at the same time (e.g., x, y, and z could each specify a call to the same piece of physical hardware, which implements a hardware-task). Thus, the specification includes a unique $NEVER$ set for every resource used by multiple tasks in the CDFG. For the sake of brevity, however, in this paper we consider the case in which we have a single $NEVER$ set specifying tasks which are run on a single microprocessor.

Example 1 As a motivational example, consider a set of control algorithms (laws) used to calculate appropriate torques for
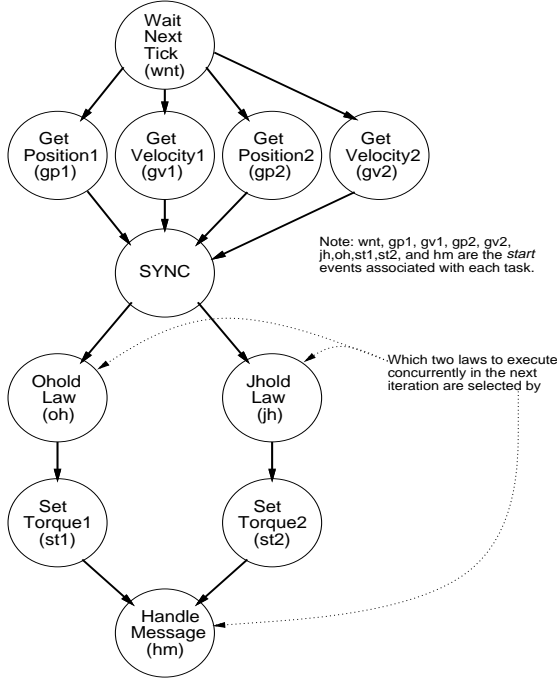
Figure 2: Robotics Example: Main Task

a robot arm. We assume that the controller manages two arms at the same time. Thus, any two of the laws (algorithms) may be selected in each execution. Note that some of the laws, e.g., Ohold2 Law, contain conditional execution of certain tasks. 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.

Figure 2 shows the overall flow of execution of the robot controller in the form of a CDFG of the main task for the system. The original specification of the main task was in Verilog. The other tasks are specified in C and Verilog. □

Worst-Case Execution Time (WCET) for individual tasks is calculated as follows. Software-tasks are compiled and input to CINDERELLA-M[2, 17], which outputs a *WCET* for each software-task, where we assume the software-task is run on a MIPS R4000 processor. Similarly, the Synopsys Behavioral Compiler^{TM} (BC^{TM})[9] generates an exact execution time for each hardware-task, which we take as a *WCET* for the hardware-tasks. These values are used to annotate the vertices in the CDFG of the system specification. Figure 3 shows a sample CDFG and a corresponding table with the *WCET* annotations.

3 Path-Based Edge Activation

We first give a motivation for the use of *path-based edge activation* to serialize tasks in a *NEVER* set. The basic point in the following example is that no static order can minimize *WCET* for all possible paths through the CDFG.

Example 2 As an example, consider Figure 3. This represents a subset of the tasks in our robot control algorithm. The
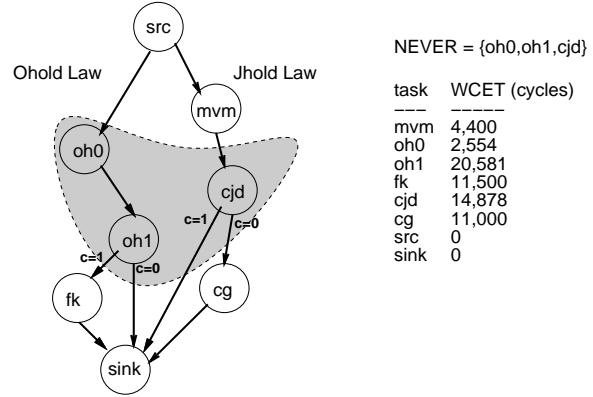

Figure 3: Dynamic Control Example

task	WCET (cycles)
mvm	4,400
oh0	2,554
oh1	20,581
fk	11,500
cjd	14,878
cg	11,000
src	0
sink	0

NEVER = {oh0,oh1,cjd}

WCET times for the individual tasks have already been calculated by CINDERELLA-M and BC^{TM}. Three tasks are specified in Verilog: mvm, fk, and cg, corresponding to matrix vector multiply, forward kinematics, and calc gravity, respectively, in Figure 1. Similarly, three tasks are specified in C: oh0, oh1, and cjd, where cjd corresponds to calc joint dynamics in Figure 1 and both oh0 and oh1 are coarser-grained groupings of tasks called by Ohold Law in Figure 1. Since our target architecture for this example contains only one microprocessor, all three software-tasks are put into a single *NEVER* set which states that their execution times cannot overlap at all. Thus, the tasks must be serialized.

Consider the *NEVER* set shaded in Figure 3. Control value c is calculated at the beginning of each iteration. Figure 4 shows the two possible paths: one path for the case $c = 1$ and a different path for the case $c = 0$. *Notice that no static order for the three tasks yields a minimum WCET in both cases.* In particular, for case $c = 1$, the order oh0, oh1 and then cjd last yields a minimal *WCET* of 38,013 cycles. However, for $c = 0$, the order oh0, cjd and then oh1 last yields the minimal *WCET* of 38,013 cycles.

The new edges needed to enforce each order is shown by the dotted arrows in Figure 4.
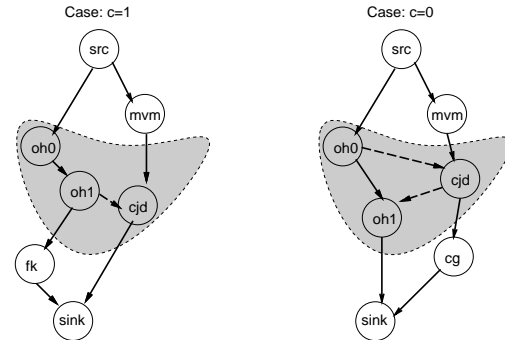

Figure 4: Dynamic Control Example: New Edges

If we were forced to select only one order for both cases, the best static order we could pick would be the order (oh0, oh1, cjd) as shown for case $c = 1$; however, for the case $c = 0$, the order (oh0, oh1, cjd) yields a *WCET* of 49,013 cycles. □

Example 2 shows that, if control information can alter the flow of task execution, then a static order may not

be able to minimize $WCET$ for all control paths. A dynamic order, selected at run-time, will perform better in general. For this reason we introduce *path-based edge activation*.

3.1 Path-Based Edge Activation

In *path-based edge activation* we add *conditional edges* to the original CDFG of the system. A *conditional edge* is an edge which is active only if a particular path, corresponding to a particular set of values for the conditional choices in the CDFG, is chosen. The addition of conditional edges changes the semantics of the task graph since tasks will not have to *wait* on the edges which are not *activated*. This is not a problem since we have central knowledge of the system.

Example 3 In Figure 4, we found different edges to implement different optimal orders for the two possible values of c (in this case the value of c is set by oh0).
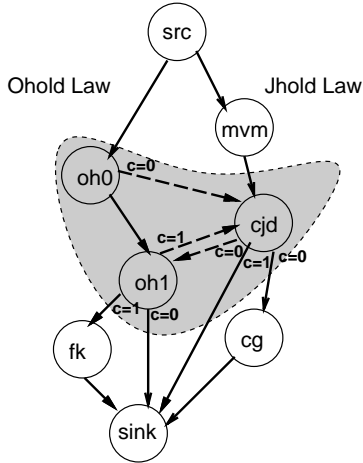

Figure 5: Dynamic Control Example: Conditional Edges

Figure 5 shows an example of a CDFG with conditional edges added. The dotted conditional edges are valid **only** if the respective conditions are true. For example, if $c = 0$, then cjd begins right after oh0 finishes (the conditional edge from oh1 to cjd is not in effect). On the other hand, if $c = 1$, then the conditional edge from oh1 to cjd is active, and so cjd begins after oh1 finishes.

In this way the path-based *conditional edges* allow the single CDFG of Figure 5 to implement both CDFGs shown in Figure 4. In particular, note the the order of execution of oh1 and cjd is dynamically changed based on the value of c at run-time, thus enabling a minimal $WCET$ of 38,013 cycles to be achieved. □

Note that so far we have made no claims about the optimality achievable with conditional edges. Clearly, a conditional edge can only be added to the CDFG in a location where we can guarantee that the conditional used to control activation of the edge is already calculated. In this paper we do not address the issue of guaranteeing that all conditionals used to control edge activation are ready in time; instead, we assume that all conditionals

controlling alternative executions in the CDFG are calculated by the beginning of each iteration of the CDFG.

4 Run-Time Scheduler Implementation

We implement our run-time scheduler as follows. From the main task, we extract a CDFG specifying the control-flow of tasks. Since we assume that the main task in invoked at a fixed rate, the CDFG we obtain is also invoked at a fixed rate. The CDFG has an equivalent Control-Flow Expression[22] which is used to synthesize a hardware FSM[16, 18]. This FSM implements the overall system control and can predictably meet the relative timing constraints, if satisfiable, specified in exact numbers of cycles between the start times of tasks, which we hypothesize cannot be satisfied by software.

4.1 Task Execution

Task execution is described in [16, 18, 19]. Briefly, we associate a *start* and a *done* event with each 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. 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 run-time scheduler hardware FSM, synthesized to implement the control-flow of the CDFG, updates the *start* vector in software as follows. First, it updates a local register containing the *start* vector. Then it triggers an interrupt on the CPU. The CPU Interrupt Service Routine (ISR) reads the register using a memory-mapped I/O read and places it into the software copy of the *start* vector. When a software-task is finished executing, it updates the *done* vector by writing the value out with memory mapped I/O. Thus, the the *done* vector in the run-time scheduler in hardware is updated.

Note that we do not want to have a single order for all software-tasks, but instead want to be able to dynamically order the software-tasks based on the values of conditionals. Therefore, we do not want to use a static priority scheduler to manage the software-tasks, as we did in [17, 18, 19]. Instead, we want to have the hardware FSM indicate which software-task to execute next. In [16] we called this *hardware-driven software execution*. Specifically, the following occurs in sequence: (1) a hardware interrupt updates the *start* vector, (2) the software Interrupt Service Routine (ISR) executes a jump to the software-task indicated by the *start* vector, and (3) when the software-task finishes, the *done* vector is written and the ISR finishes.

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 scheduler that executes different threads (software-tasks) based on the *start* vector.

We have described the synthesis of the hardware executive manager in [16, 22, 18, 19]. In this paper we focus on the generation of *conditional edges* which allow our run-time scheduler to implement *path-based edge activation* and thereby make *WCET* as small as possible.

5 Real-Time Analysis

We aim to predictably satisfy real-time constraints in the form of control-flow (precedence and alternative execution) constraints, resource constraints, relative timing constraints, and a rate constraint. We assume that we have as input a CDFG, a rate constraint on the graph, and a *NEVER* set specifying a resource constraint on software-tasks. The formulation shown here does not include *NEVER* sets of hardware-tasks (hardware resource constraints) for the sake of simplicity.

To predictably satisfy the rate constraint, we need a *worst case execution time (WCET)* for each task and a *WCET* for the control-flow of the set of tasks under the rate constraint. We obtain the *WCET* times for the individual tasks from CINDERELLA-M and BC^{TM}[9]. We need some assumptions to compute the *WCET* for the set of tasks.

Assumption 5.1 *We have a control/data-flow graph (CDFG) representing the set of tasks under the rate constraint, a WCET for each task, and a NEVER set specifying tasks that must be executed in a mutually exclusive manner. The CDFG includes alternative execution (if/case statements) but does not include loops.*

The use of *NEVER* sets to provide mutual exclusion for hardware-tasks is covered in [22]. We consider here only a single *NEVER* set of software-tasks executed on the same CPU.

Assumption 5.2 *Each task, once started, runs to completion.*

Assumption 5.3 *Hardware-software communication time is included in the WCET of each task and/or is included as a distinct task.*

We have several communication primitives, such as shared memory and FIFOs, with interface generation along the lines of [14, 15].

Assumption 5.4 *Interrupts that initiate task execution come only from the hardware run-time scheduler as described in Section 4.1.*

An example of a CDFG and resulting scheduling options were shown in Examples 2 and 3. Example 2 shows a difficult problem in that a *NEVER* set of software-tasks may cross parallel paths. We cannot use one execution of a longest path algorithm to solve this problem

because the execution time of each node in a *NEVER* set depends upon the scheduling of the other nodes in the *NEVER* set. We could enumerate all the possible orderings of nodes in the *NEVER* set and then execute a longest path algorithm for each permutation. However, we would then perform many redundant calculations. This problem can be shown to be NP-Complete using the Resource Constrained Scheduling NP-Complete problem of [20].

In this paper, we formalize the approach to finding the schedule shown in Example 3.

5.1 CLARA2 Real-Time Analysis

We want to find an exact scheduling of the tasks, with a *NEVER* set containing all the software-tasks, where the other tasks are all hardware-tasks. So we design an algorithm to suit this specific problem.

We take as input both the CDFG annotated with *WCETs* for each task and a *NEVER* set specifying the mutually exclusive tasks. Similar to [24], we then consider all possible paths.

For a given set of values of the conditionals, a particular path through the CDFG is defined (note that we assume that all conditionals controlling alternative executions are calculated by the beginning of each iteration of the CDFG). For this set of values of conditionals, we then add *conditional edges* to minimize the *WCET* for that path. We solve this subproblem using the *constructive heuristic scheduling* of [18, 19], which solves the problem for a *Directed Acyclic Graph (DAG)*. Note that for a given path defined by a set of values of the conditionals, the CDFG reduces to a DAG.

```
Solve_order(CDFG, NEVER);
    foreach path determined by a unique set of conditional values
    begin
        DAG = subset of CDFG determined by path
        Schedule DAG using constructive_hueristic of [18, 19]
        Add conditional edges to enforce DAG schedule
    end
endmodule
```

Figure 6: CLARA2 Dynamic Scheduling Algorithm

The psuedo-code for the CLARA2 Scheduling Algorithm is shown in Figure 6.

Note that this is exponential in the worst case, since the number of distinct paths may be exponential. Nevertheless, we conjecture that this approach will prove to be reasonable for a large set of real-time hardware/software scheduling problems.

Thus we have a dynamic order which, given our assumptions, chooses a small *WCET* while satisfying, dynamically, mutual exclusion of tasks in the *NEVER* set.

This final output is an upper bound on the *WCET* of the graph given the dynamic order of execution of software-tasks in the same *NEVER* set.

5

So we now can analyze satisfiability of a rate constraint in a dynamically changing, concurrent execution of hardware-tasks and software-tasks, given our run-time scheduler implementation.
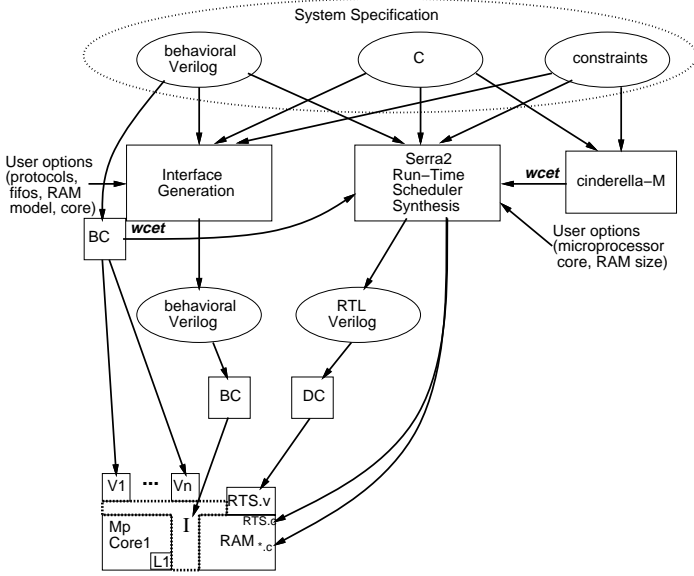
6   Tool Flow and Target Architecture

Figure 7: Tool Flow and Target Architecture

Figure 7 shows our tool flow, where BC labels the Synopsys Behavioral Compiler^{TM}[9] and DC labels the Synopsys Design Compiler^{TM}. Hardware-tasks are specified in Verilog and 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 implementation of a synthesized system can vary from a system on a chip to a board or set of interconnected components.

Constraints include rate constraints, relative timing constraints (minimum and maximum separation), and software resource constraints. Precedence constraints and control-flow constraints (alternative execution) are implicit in the task specification.

The system-level tasks in Verilog and C, as well as constraints, are input to SERRA2 and to a tool that generates the interface. One of the tasks is specified as the main task. CINDERELLA-M, which we have ported to the MIPS R4K, takes input in C and outputs a *worst-case execution time (WCET)* for each software-task (note that bounds on loops must be provided by the user)[2]. Similarly, BC^{TM} generates an exact execution time for each hardware-task, which we take as a *WCET* (loop bounds must be provided here in some cases as well). When comparing BC^{TM}-generated *WCETs* with software *WCET*, we convert all delays to the number of microprocessor clock cycles (since the hardware clock speed is typically slower.)
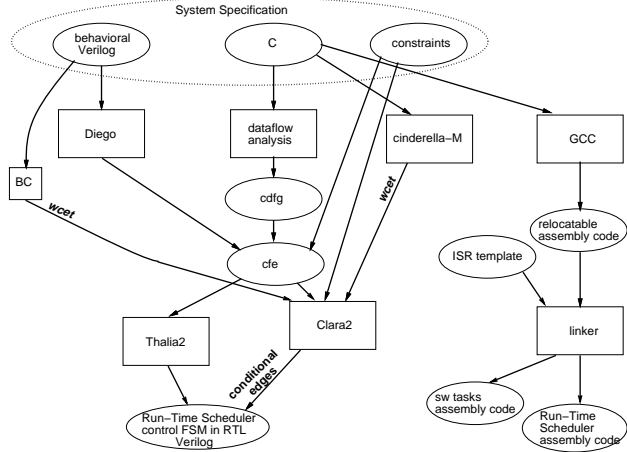

Figure 8: Block diagram of SERRA2: the boxes indicate tools and the ovals indicate data.

6.1   SERRA2 Run-Time Scheduler Synthesis

The flow of the SERRA2 Run-Time Scheduler Synthesis tool is shown in Figure 8. SERRA2 synthesizes the control-unit of the scheduler into a hardware FSM and generates the ISR-based scheduler of the software-tasks.

For the software that runs on the microprocessor core (CPU), the individual software-tasks are compiled together with the ISR standard C compilers and linkers. Memory-mapped I/O is called with C pointers set explicitly to the appropriate addresses.

Data and program memory are statically allocated. The ISR, which is the interrupt handling portion of the run-time scheduler, reads in a *start* vector that specifies which task is now ready to be executed in software.

We 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 can be executed.

7   Example and Experimental Results

For our example, we consider the code of Figures 1 and 2. Jhold Law and Ohold Law of are implemented with the hardware- and software-tasks shown in Figure 3.

We performed real-time analysis using the CLARA2 tool which is a modified version of the CLARA tool [18, 19]. The order of execution for the software-tasks is decided at run-time based on the values of conditionals. This provides for the upper bound on execution speed for the application under worst-case conditions.

The system begins each iteration once a millisecond. After obtaining the positions and velocities of the two robot arms, the run-time scheduler starts the execution of *mvm* in hardware for Jhold Law and *oh*0 in software

6

for `Ohold Law`. It continues with interleaved hardware-software execution as shown in Figures 4 and 5 based on the value of conditional c. In this way CLARA2 achieves a $WCET$ of 38,013 cycles. Note that if limited to a single static order for the software-tasks (as is the case in the original CLARA), the minimum $WCET$ possible for this example would be 49,013 cycles; thus, conditional edges allow for a 22% decrease in $WCET$ in this case.

Software-Task	Lines C	Lines Assem.	$BCET$	$WCET$
cjd	286	1177	9,989	14,878
oh0	90	237	1,598	2,554
oh1	693	3263	12,424	20,581
int-ser-routine	N/A	26	11	20

Table 1: Code space, $BCET$ and $WCET$ for sw-tasks.

Hardware-Task	Lines Verilog	Area	$BCET$	$WCET$
mvm	629	33,645	4,400	4,400
fk	2362	42,168	11,500	11,500
cg	2897	59,587	11,000	11,000
run-time-sch-hw	484	413	N/A	99,701

Table 2: Results for the synthesis of hw-tasks.

Table 1 presents the results for the compilation of the software and best- and worst-case execution time estimation with CINDERELLA-M. In Table 2, we see the results for the synthesis of the hardware tasks of Figure 3 using the Behavioral Compiler^{TM}, except for the run-time scheduler hardware part which was synthesized with the Design Compiler^{TM}. The third column in Table 2 shows the number of gate equivalents the hardware required using the LSI 10K Logic library. We clock the hardware at 10 MHz. Using a MIPS R4K model in Verilog, we simulated the Robot Arm Controller in Verilog using Chronologic's VCS^{TM}.

8 Conclusion

We have shown how to handle additional control-flow constraints during real-time analysis in hardware-software codesign with a custom run-time system. The CLARA2 Real-Time Analysis tool, which is embedded in the SERRA2 Run-Time Scheduler tool, helps designers perform system-level design with hardware and software at a coarse level of granularity. We can predictably meet hard real-time constraints with our approach, based on dynamic hardware-driven execution of software tasks, with results significantly superior to previously reported results.. The final result is tighter execution bounds thus squeezing more performance out of the same components than with a traditional RTOS and associated real-time analysis.

References

[1] P. Eles, K. Kucheinski, Z. Peng and A. Doboli, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems," Proceedings of the Design, Automation and Test in Europe Conference, pp. 132-138, February 1998.

[2] S. Malik, W. Wolf, A. Wolf, Y. Li, and T. Yen, "Performance Analysis of Embedded Systems," in G. De Micheli and M. Sami, editors, Hardware/Software Co-Design, pp. 45-74, Kluwer Academic Publishers, Norwell, MA, 1996.

[3] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real time environment," Journal of the ACM, 20(1):46-61, January 1973.

[4] L. Sha, R. Rajkumar and S. Sathaye, "Generalized rate monotonic scheduling theory: a framework for developing real-time systems," Proceedings of the IEEE, 82(1):68-82, January 1994.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell and A. J. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," Software Engineering Journal, pp. 284-292, September 1993.

[6] N. Audsley, A. Burns, R. Davis, K. Tindell and A. J. Wellings, "Fixed Priority Pre-emptive scheduling: A Historical Perspective," Real-Time Systems, (8):173-198, 1995.

[7] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," IEEE Proceedings on Parallel and Distributed Systems, 6(4):412-420, April 1995.

[8] T. Yen and W. Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems," Proceedings of International Conference on Computer Design, pp. 64-69, 1995.

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

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

[11] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann and M. Trawny, "The COSYMA environment for hardware/software cosynthesis of small embedded systems," Microprocessors and Microsystems, 20 (1996) pp. 159-166.

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

[13] Pai H. Chou and Gaetano Borriello, "Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems," Proceedings of the 31st Design Automation Conference, pp. 1-4, June 1994.

[14] 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.

[15] 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.

[16] V. Mooney, T. Sakamoto, G. De Micheli, "Run-Time Scheduler Synthesis For Hardware-Software Systems and Application to Robot Control Design," 5th. Int'l Workshop on Hardware/Software Codesign,, pp. 95-99, Braunschweig, Germany, March 1997.

[17] V. Mooney and G. De Micheli, "Real Time Analysis and Priority Scheduler Generation for Hardware-Software Systems with a Synthesized Run-Time System," Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD'97), 605-612, 1997.

[18] V. Mooney, Hardware/Software Co-Design of Run-Time Systems, Ph.D. Thesis, Technical Report CSL-TR-98-762, http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs%2fCSL-TR-98-762, Stanford, CA, June, 1998.

[19] V. Mooney and G. De Micheli, Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems, Design Automation of Embedded Systems, to appear.

[20] M. Garey and D. Johnson, Computers and Intractability A Guide to the Theory of NP-Completenes, W. H. Freeman and Company, N.Y., 1979, pg. 239.

[21] T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, 1990, pg. 35.

[22] 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.

[23] C. N. Coelho Jr., Analysis and Synthesis of Concurrent Digital Systems Using Control-Flow Expressions, Ph.D. Thesis, Technical Report CSL-TR-96-690, http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs%2fCSL-TR-96-690, Stanford, CA, March, 1996.

[24] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Transactions on CAD/ICAS, Vol. 10, No. 1, January, 1991.