Real Time Analysis and Priority Scheduler Generation for Hardware-Software Systems with a Synthesized Run-Time System

Vincent J. Mooney III & Giovanni De Micheli
Computer Systems Laboratory, Stanford University
Gates Computer Science Building
Stanford, CA 94305
mooney@pampulha.stanford.edu, nanni@galileo.stanford.edu

Abstract

We present a tool that performs real-time analysis and priority assignment for software tasks in a mixed hardware-software system with a custom run-time scheduler. The tasks in hardware and software have precedence constraints, resource constraints, relative timing constraints, and a rate constraint. A dynamic programming formulation assigns the static priorities such that a hard real-time rate constraint can be predictably met.

We describe the task control/data-flow extraction, run-time scheduler implementation, real-time analysis and priority scheduler template. 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

Designers of real-time embedded systems often have timing constraints that they must meet for the design to be successful. To support soft and hard real-time constraints, system designers need tight bounds on execution delays. In hardware-software codesign, scheduling resources to meet these tight bounds is a critical problem because there may be parallel threads of execution in the application with the same resource required by different threads. In this paper, we consider the following formulation of the real-time analysis problem : we represent the system with a single graph where the nodes represent software or hardware, the graph edges represent dependencies (precedence constraints), 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 [19].

Previous approaches to real time analysis have focused on software [1] 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[10].

Our formulation is similar to [7, 10, 15]. However, in our case we synthesize a custom run-time scheduler in hardware and software for the application [19]. 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 for capture of a satellite in space) our approach can provide precise real-time bounds.

In our approach, if a solution is found, we output the task priorities and guarantee that the system meets its relative timing constraints and its rate constraint, assuming the system uses a custom run-time scheduler. While the approach of [8, 9] is more general for verification purposes after the priority for each task is assigned, it does not address the issue of finding optimal priorities.

The rest of the paper is organized as follows. Section 2 explains our design approach and corresponding requirements. Section 3 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 4 describes our implementation of the run-time scheduler, including the priority scheduler template. Section 5 presents the real-time analysis and priority generation for software tasks. Section 6 describes the tool flow and target architecture. Section 7 gives some experimental results and presents an example from robotics. Finally, Section 8 concludes the paper.

2 Motivation

System-level design requires a division of the system application into tasks which coordinate with each other. We call a task implemented in hardware a *hardware-task*; a task implemented in software is called a *software-task*. A *coarse-grained* partition specifies these tasks and their overall flow. We consider the system application after a designer or a partitioning tool has split the application into hardware-tasks and software-tasks.

We assume that the system requires some static scheduling, especially in the coordination of hardware-tasks, as well as dynamic scheduling, given the inexact delay of software and the randomness of the stimuli coming from the envi-

1092-3152/97 $10.00 © 1997 IEEE

ronment. The run-time scheduler synthesis of [19] supports the execution of software-tasks through an interrupt triggering mechanism where the hardware communicates to a software scheduler which software-tasks are ready to execute. However, the software scheduler was implemented by hand; in this paper, we implement the software portion of the run-time scheduler with a preemptive fixed priority scheduler. Our tool, called CLARA, automates the generation of priorities for the software-tasks, as well as *worst case execution time (wcet)* calculation for subsets of hardware- and software-tasks under a hard real-time rate constraint.

A typical application domain for such a design style is embedded systems. We assume the existence of mature high-level synthesis tools and software compilers, as well as the availability of processor cores. A typical task size is 50 to 200 lines of Verilog or C; the only limit on task size, however, comes from the high-level synthesis tool or compiler chosen. We assume that tasks are custom written 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.

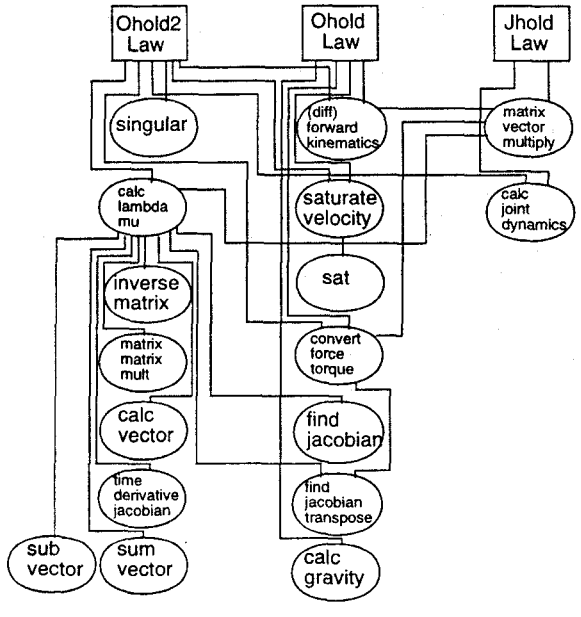


Figure 1: Robotics Example: Concurrent Control Laws

3 TaskSIF Extraction

The input specification is a collection of tasks written 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 each task we extract a Control/Data-Flow Graph (CDFG) of the tasks it invokes, where each node in the CDFG corresponds to a call to another task. If a task does not call any other task, then it has no such CDFG. We call this kind of task a *leaf task*. A task which is not the main task nor a leaf task is an *intermediate task*. An intermediate task must trace back its invocation to the main task, and the intermediate task must itself invoke at least one leaf task. We assume that an intermediate task has all computation specified in leaf tasks. If an intermediate task does contain some computations, a new leaf task can be generated containing these computations. This allows us to flatten the hierarchical description and generate a CDFG of the system where all nodes are leaf tasks. We represent the flattened CDFG in ASCII in the Task Sequencing Intermediate Format (TaskSIF). We assume that we have a rate constraint specified for the TaskSIF graph of the system. In other words, we assume that the main task is invoked at a fixed rate.

We support the specification of tasks that cannot execute concurrently through the use of $NEVER$ sets. In general, $NEVER$ sets can model mutual exclusion; here, we use $NEVER$ sets to model resource constraints. For example, $NEVER = \{a, b, c\}$ indicates that tasks a, b, and c can never be active at that same time. We make use of this feature to specify resource constraints such as **(i)** multiple calls to the same piece of physical hardware (which implements a hardware-task), or **(ii)** software-tasks executed on the same microprocessor. In general, you can have multiple $NEVER$ sets. In this paper, however, we consider the case in which we have a single $NEVER$ set which we use to serialize the software-tasks executed on the same CPU. Similarly, tasks that must begin execution at the same time are specified through the use of $ALWAYS$ sets; e.g. $ALWAYS = \{a, b, c\}$ indicates that tasks a, b, and c must each begin execution at the same time. Note that this is the same as having bilateral relative timing constraints of zero weight, which our run-time scheduler also supports[19]. Thus, we do not consider $ALWAYS$ sets explicitly in the formulation of our problem.

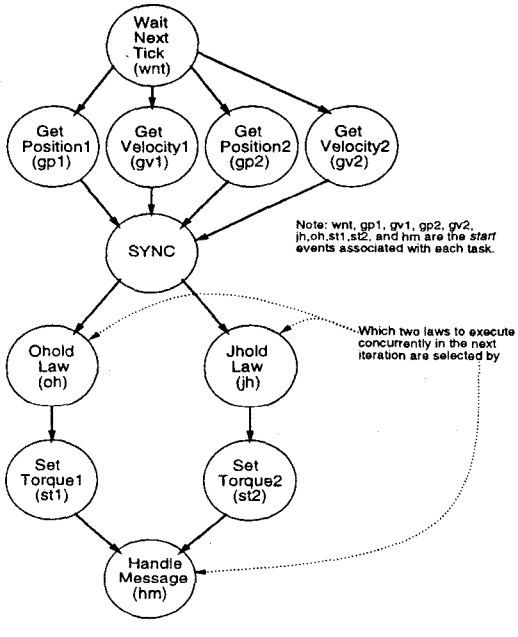


Figure 2: Robotics Example: Main Task

Example 1 As a motivational example, consider a set of control algorithms (laws) used to calculate appropriate torques for 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. 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 intermediate tasks which call other intermediate tasks and leaf tasks in a particular sequence.

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.

Note that Figure 2 must complete once every millisecond. Thus, we have a rate constraint on the graph. □

We wrote a new backend for CINDERELLA[1] for MIPS assembly run on a MIPS R4K processor; we call the new tool CINDERELLA-M. The leaf software-tasks are compiled and input to CINDERELLA-M, which outputs a *worst-case execution time (wcet)* for each task. Similarly, the Synopsys Behavioral Compiler^{TM} (BC^{TM})[11] 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 leaf nodes in the final TaskSIF graph of the system specification. Figure 3 will show a sample TaskSIF graph and a corresponding table with the *wcet* annotations.

4 Run-Time Scheduler Implementation

We implement our run-time scheduler as follows. Starting from the hierarchical CDFG specified by the main task, intermediate task(s), and leaf tasks, we flatten the description until we have a single TaskSIF graph. Since we assume that the main task in invoked at a fixed rate, the TaskSIF graph we obtain is also invoked at a fixed rate. The TaskSIF graph has an equivalent Control-Flow Expression[23] which is used to synthesize a hardware FSM[19]. 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 [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 TaskSIF graph, 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.

The *start* vector may specify that several software tasks are ready. Thus, we generate a preemptive static priority scheduler which executes the highest priority software-task among the tasks indicated by the hardware FSM as ready to execute. The priority based scheduler is always called by the ISR after fetching the new *start* vector into memory.

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 preemptive static priority scheduler that executes different threads based on eligible software-tasks as indicated by the *start* vector.

We have described the synthesis of the hardware executive manager in [19, 23]. In this paper we focus on the generation of a priority scheduler for the software-tasks.

4.2 Priority Scheduler Template for Software

A task can be in one of four distinct states: *ready*, *running*, *suspended*, or *terminated*. We accomplish this with a single *ready* list, implemented as a linked list. The *running* task is at the top of the *ready* list. *Ready* tasks are below the *running* task in the order of their priorities. A *suspended* task is pushed down the *ready* list according to its priority, with the higher priority task which caused its suspension on top. A *terminated* task does no appear on the list. (Note that we eliminate the *delay* list traditionally used in priority schedulers, e.g. as described in [18].) The register file that contains the process state information is saved only when a task is suspended (i.e. we eliminate context switching when one task ends and another begins, in which case there is no need to save the register file). In operating systems terms, the run-time scheduler software portion implements priority-based job scheduling (multiprogramming). Strictly speaking, this is not multitasking since there is no time-shared access to CPU compute cycles.

Clearly, for this implementation to work, we need a priority for each software-task. We obtain the priorities from the real time analysis.

5 Real Time Analysis

We aim to predictably satisfy real-time constraints in the form of control-flow (precedence) constraints, resource constraints, relative timing constraints, and a rate constraint. We assume that we have as input a TaskSIF, 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}[11]. We need some assump-

tions to compute the *wcet* for the set of tasks.

Assumption 5.1 *We have a directed acyclic graph (TaskSIF) 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 use of *NEVER* sets to provide mutual exclusion for hardware-tasks is covered in [23]. 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.*

We will relax this assumption later when calculating *wcet* involving software-tasks which can be partially executed before being interrupted.

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 [16, 17].

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

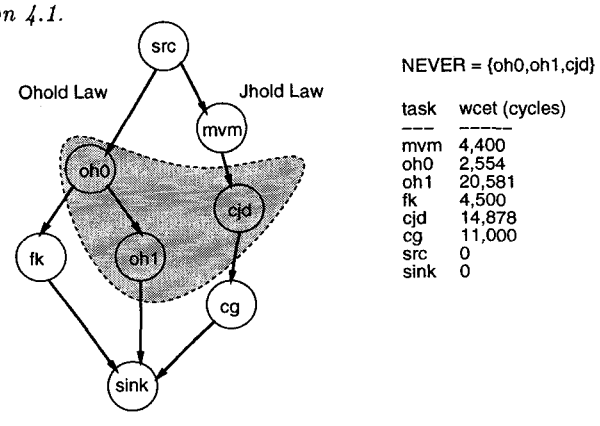


Figure 3: Dynamic Programming Example

Example 2 As an example, consider Figure 3. This represents a subset of the tasks in our robot control algorithm. The *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`, `(diff) 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. A first-come-first-serve scheduling algorithm would schedule `oh0` first, then `oh1` (since `mvm` is still executing when `oh0` finishes), and `cjd` last, resulting in a *wcet* of 49,013 cycles for the graph. However, if `oh1` were executed **after** `cjd`, the *wcet* would be 39,859 for the graph. □

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 [2]. We could enumerate all the possible orderings 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].

5.1 Dynamic Programming

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 find an optimal ordering of the software-tasks with a dynamic programming formulation of the problem. Dynamic programming is an exact solution method that in our application is above polynomial in the worst case. However, due to the coarse granularity of the model, the number of tasks is usually not large and dynamic programming can be executed quickly. Dynamic programming allows us to take into account the control-flow constraints and store subcalculations, thus reducing the number of computations. A good overview of dynamic programming is contained in [22].

5.1.1 Dynamic Programming Formulation

We take as input both the TaskSIF graph annotated with *wcets* for each leaf task and a *NEVER* set specifying the mutually exclusive tasks. We divide the problem into stages according to the number of elements in the *NEVER* set. We use the following definitions:

Definition 5.1 *Let there be n stages, where in each stage we decide which among n tasks to schedule.*

The number of stages n is set equal to the number of nodes in the *NEVER* set plus two (for the source and the sink).

Definition 5.2 *Let state s in stage i denote the current task ready to start execution and the subsequent tasks executed in stages* $\{i+1,\ldots,n\}$.

We name state s with the name of the current task ready to start execution. Note that given an ordering of software-tasks, the rest of the graph is scheduled with an ASAP schedule. Since dynamic programming stores all optimal states, i.e. sets of task schedules for stages $\{i, i+1,\ldots,n\}$, the Markov property necessary for dynamic programming to produce an optimal solution holds [22].

Definition 5.3 *Let the decision variables* x_i, $i \in \{1,2,\ldots,n-1\}$ *denote the task scheduled to occur next, i.e. after stage i.*

Note that the decision variable in Operations Research literature is not restricted to binary values. In this case, x_i takes on symbolic values, i.e. task names.

Example 3 For Figure 3 we have $n = 5$: tasks under consideration are `src`, `oh0`, `oh1`, `cjd`, and `sink`. Since the `sink` is always executed last, $x_{n-1} = x_4 =$ `sink`. The possible tasks executed before the `sink`, and thus in stage 4, are $s =$ `oh1` and $s =$ `cjd`. □

Definition 5.4 *Let* $f_i(s,x_i)$, $i \in \{1,2,\ldots,n-1\}$, *be the worst case execution time for stages* $\{i, i+1,\ldots,n\}$, *given that the first task in s is executed in stage i and task* x_i *is*

executed in stage $i+1$. $f_n(s)$ is defined to be zero since there is no task to execute after the last stage, and the last task executed is always the **sink**, *which takes zero cycles.*
Recall that tasks not in the $NEVER$ set are all hardware-tasks and are scheduled ASAP.

Definition 5.5 *Let $f_i^*(s)$, $i \in \{1,2,\ldots,n-1\}$, be the corresponding minimum value of $f_i(s,x_i)$ over all possible x_i.*

Definition 5.6 *Given state s and stage i, let x_i^* denote the value of x_i that minimizes $f_i(s,x_i)$.*

Example 4 Continuing with Figure 3, $f_5(s)$ is defined to be zero, and, as in Example 3, x_4 = sink. Then $f_4(oh1, x_4) = 20,581$ and $f_4(\texttt{cjd}, x_4) = 25,878$. Since there is only one possibility for x_4, $f_4^*(\texttt{oh1}) = 20,581$ and $f_4^*(\texttt{cjd}) = 25,878$. Figure 4 shows the two sets of nodes scheduled and their *wcet* paths. □

Definition 5.7 *Given state s, task x_i to execute next, and $wcet_{succ}$ = wcet of the successors of task s, let*

$$wcet_{extra} = \begin{cases} wcet_{succ} - wcet_{x_i} & \text{if } wcet_{succ} > wcet_{x_i} \\ 0 & \text{otherwise} \end{cases}$$

and let $wcet_{sx_i} = wcet_s + wcet_{extra}$.

In calculating $wcet_{succ}$, we schedule the subgraph covered by the successors of task s using an ASAP schedule (a worst-case polynomial time computation). If we find a successor that is in the $NEVER$ set, then we use $wcet_{x_j}$, where x_j is the decision variable, already scheduled, corresponding to the successor (if no such x_j exists, then we disallow this order of tasks).

Thus we have the following:

$$f_i^*(s) = \min_{x_i} f_i(s,x_i) = f_i(s,x_i^*), i \in \{1,2,\ldots,n-1\}$$

where $f_i(s,x_i) = wcet_{sx_i} + f_{i+1}^*(x_i)$.

5.1.2 Dynamic Programming Solution

The number of stages n is set equal to the number of nodes in the $NEVER$ set(s) plus two (for the source and the sink). Our approach starts with the last stage, stage n, and progressively works its way back to the first stage. We set the last stage to be the sink and the first stage to be the source (we always have a source and a sink according to Assumption 5.1).

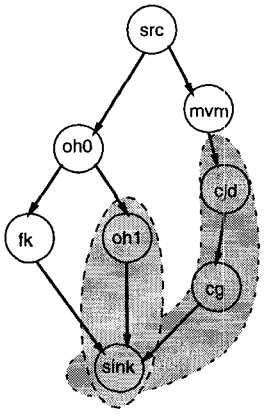
In order to begin with the last stage, we schedule the sink, yielding $f_n^*(s) = 0$.

For stage $n-1$, the *wcet* is determined entirely by the current state (whichever task is chosen to execute). Therefore, our dynamic programming table need only include s, $f_{n-1}^*(s)$, and x_{n-1}^*.

s	$f_4^*(s)$	x_4^*
oh0	-	sink
oh1	20,581	sink
cjd	25,878	sink

Table 1: Dynamic Programming Example Stage $n-1$

Example 5 Consider Figure 3. We have $n = 5$ stages. For stage 5 we found that $f_n^*(s) = f_5^*(s) = 0$. Table 1 shows the calculations for stage 4. Note one optimization already: oh0 is not schedulable in this stage due to control-flow (precedence) constraints.



NEVER = {oh0,oh1,cjd}

s	$f_4^*(s)$
oh1	20,581
cjd	25,878

Figure 4: Dynamic Programming Example Stage $n-1$

Figure 4 shows the two sets of nodes scheduled and their *wcet* paths in this pass of the algorithm. □

For stages $n-2$ through 2, we calculate a table of values containing each possible task that can be scheduled in that stage and each possible task than can be scheduled in the next stage. In the worst case, the table has $(n-2)^2 - n$ entries. Note that for each entry, in the worst case $|V|$ operations have to be performed in calculating $wcet_{sx_i}$, where V denotes the vertices in the task-CDFG graph.

x_3	$f_3(s,x_3) = wcet_{sx_3} + f_4^*(x_3)$		$f_3^*(s)$	x_3^*
s	oh1	cjd		
oh0	23,135	-	23,135	oh1
oh1	-	46,459	46,459	cjd
cjd	35,459	-	35,459	oh1

Table 2: Dynamic Programming Example Stage $n-2$

Example 6 Continuing our attempt to optimally schedule Figure 3, we pass now to stage 3. Table 2 shows the calculations for this stage. The first entry contains the it wcet if oh0 is scheduled in stage 3 and oh1 in stage 4 (and the sink in stage 5). Note that it is not possible to schedule oh0 in stage 3 and cjd in stage 4 due to control-flow constraints, so the entry is empty. Note also that there is no column for oh0 since it was not possible to schedule it in stage 4.

To calculate the additional *wcet*, $wcet_{sx_3}$, given that we execute task s in this stage (3) and task x_3 in the next stage (4), requires scheduling the subgraph covered by task s, task x_3 and their successors. We use an ASAP schedule. □

x_2	$f_2(s,x_2) = wcet_{sx_2} + f_3^*(x_2)$			$f_2^*(s)$	x_2^*
s	oh0	oh1	cjd		
oh0	-	49,013	38,013	38,013	cjd
oh1	-	-	-	-	-
cjd	38,013	-	-	38,013	oh0

Table 3: Dynamic Programming Example Stage 2

Example 7 Next consider stage 2. Table 3 shows the calculations for this stage. Note that for state s = oh1, it does not make sense to consider x_2 = cjd since we cannot have x_1 = oh0 (oh0 must execute *before* oh1, not after). Since the other entries are similarly not possible, the entire row is empty. □

For our last computation, namely stage 1, there is only one starting state: the source. So the table has only one row.

Example 8 Now for the last set of computations, stage 1. Ta-

x_1	$f_1(s,x_1) = wcet_{sx_1} + f_2^*(x_1)$			$f_1^*(s)$	x_1^*
s	oh0	oh1	cjd		
src	39,859	-	42,413	39,859	oh0

Table 4: Dynamic Programming Example Stage 1

ble 4 shows the calculations for this stage. Note that the algorithm finally takes into account the *wcet* for task mvm, making the option of selecting cjd to execute before oh1 less favorable. □

The final optimal ordering of tasks can be found by tracing back the x_i^* vectors, starting with x_1^*.

Example 9 The optimal order for our example starts with x_1^* = oh0. The entry for oh0 in x_2^* is cjd. Finally, the entry for cjd in x_1^* is oh1. So the optimal order of execution is oh0 first, cjd second, and oh1 last. □

Thus we have the optimal order (given our assumptions) of execution of tasks in the *NEVER* set. We use this order to statically set the priorites for the software-tasks.

5.2 Calculation of *wcet*

To calculate the *wcet* of the entire graph, we use the following costs, obtained by analyzing our run-time scheduler software code executed on a MIPS R4K model: context switch = 126 cycles, interrupt overhead = 20 cycles, and priority scheduler task selection = 44 cycles.

For the interrupt, we use pin **Int(0)** on the MIPS R4K model and do not save the register set before passing control to the priority scheduler software. Otherwise our interrupt overhead would be much larger. The priority scheduler software executes a context switch only if necessary to save/restore the state of a suspended process. Note that the context switch time includes time for storing the floating point registers.

With these costs, we calculate the *wcet* of the entire graph. We use the priority scheduler with the priorities found via dynamic programming. This increases our CPU utilization by starting execution of a low priority task that is ready when no higher priority task is yet ready. In other words, we relax Assumption 5.2 and support multiprogramming. However, without Assumption 5.2, our static priorities are no longer optimal – in fact, priorities may have to by dynamic to guarantee optimality with tasks allowed to interrupt each other. However, relaxing Assumption 5.2 can allow us to reduce *wcet* for the graph, thus improving our solution.

Note that we assume no semaphores are used, and that precedence constraints are enforced by the run-time scheduler. We could support the use of semaphores in two ways. One simple way is to reinstate Assumption 5.2 and execute software-tasks only in order of priority (e.g. the task with priority 3 cannot execute until the task with priority 2 has completed, and the priority 2 task can only execute after the priority 1 task has completed). Another way is to increase the time for priority scheduler task selection to allow enough time for any task to exit a critical section; in other words, calculate a *wcet* for any task's critical section, and add that amount to the priority scheduler task selection time (the scheduler would know, of course, if a task is in a critical section and when the task has exited).

Example 10 Consider Figure 3. We use the priorities found in Example 9. We find that the run-time scheduler causes three interrupts. After the second interrupt, oh1 executes until it is suspended when task cjd is ready. After cjd finishes, the priority scheduler switches back to oh1.

A straightforward ASAP schedule is used. Several of the software tasks have loops, but none of the tasks use semaphores. In this example, the critical path is in the execution of software.

sw-task	# cycles	hw-task	# cycles
int-ser-routine	20	mvm	4,400
priority-sch-sw	44		
oh0	2,554		
int-ser-routine	20		
priority-sch-sw	44		
oh1	1,718		
int-ser-routine	20	fk	4,500
context-switch	126		
priority-sch-sw	44		
cjd	14,878		
priority-sch-sw	44	cg	11,000
context-switch	126		
oh1	18,863		

Table 5: *WCET* Calculation Example

The overall *wcet* is 38,501 cycles. □

This final output is an upper bound on the *wcet* of the graph given the priorities assigned to software-tasks in the same *NEVER* set.

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

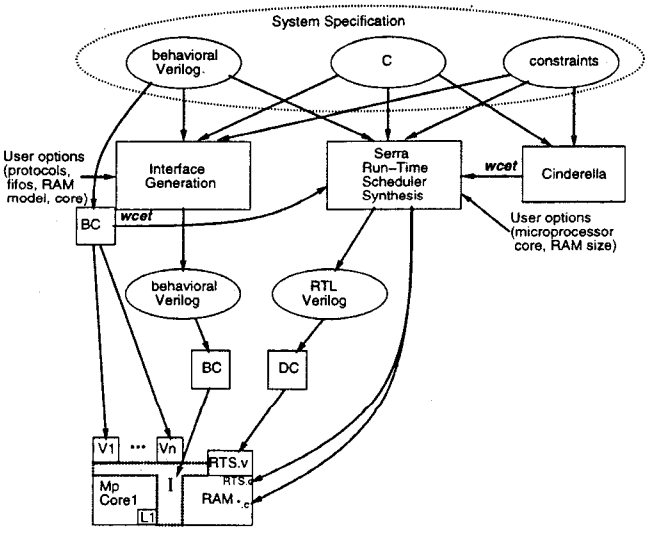


Figure 5: Tool Flow and Target Architecture

Figure 5 shows our tool flow, where BC labels the Synopsys Behavioral Compiler^{TM}[11] 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 are implicit in the task specification.

The system-level tasks in Verilog and C, as well as constraints, are input to SERRA 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)[1]. 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.)

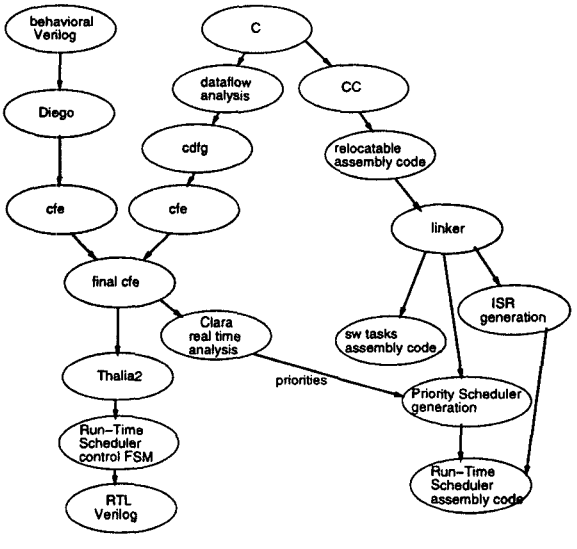


Figure 6: Block diagram of SERRA

6.1 SERRA Run-Time Scheduler Synthesis

The flow of the SERRA Run-Time Scheduler Synthesis tool is shown in Figure 6. SERRA synthesizes the control-unit of the scheduler into a hardware FSM and generates the static priorities for the software-tasks. SERRA does not generate the run-time C code but instead uses templates of the priority scheduler in C, the Interrupt Service Routine (ISR) in MIPS assembly and context switch code in MIPS assembly.

For the software that runs on the microprocessor core (CPU), the individual software-tasks are compiled together with the priority scheduler, ISR, and context switch code using 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 tasks are ready to be executed in software.

We end up with a set of software-tasks and their start addresses in the program code. Thus, we have a table of software-tasks and their entry points as seen in Table 6.

Entry	Value
0	Pointer to sw-task 0
1	Pointer to sw-task 1
.	...
n	Pointer to sw-task n

Table 6: Entry Table for Software-Tasks

Therefore, given a particular value of the *start* vector, the appropriate software-task(s) can be executed by the priority scheduler.

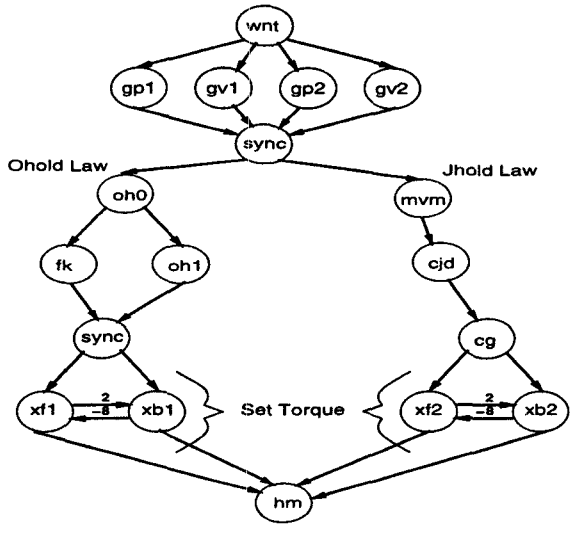


Figure 7: TaskSIF graph of Robot Arm Controller

7 Example and Experimental Results

For our example, we consider the code of Figure 2. `Jhold Law` and `Ohold Law` of Figure 2 are implemented with the subtasks shown in Figure 3. The TaskSIF graph, including the leaf tasks that implement `Set Torque`, are shown in Figure 7. Note that `Xmit Frame1` ($xf1$) and `Xmit Bit1` ($xb1$) of `Set Torque1` have a strict relative timing constraint of $xb1$ starting no less than 2 cycles after $xf1$ and no more than 8 cycles after. The exact same constraint holds for `Set Torque2`. This constraint could not always be satisfied with control signals generated by a run-time scheduler in software (note in Figure 5 we have an L1 cache).

We perform real-time analysis using the CLARA tool which has been implemented in 14,000 lines of C. The optimal order of execution for the software-tasks is passed to the priority scheduler. This provides for the upper bound on execution speed for the code 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 *oh0* in software for `Ohold Law`. It

continues with interleaved hardware-software execution as shown in Table 5. Finally, it tightly schedules accesses to `Xmit Frame` and `Xmit Bit` to set the torques for the robot.

The scheduling of tasks shown in Figure 7 but not in Figure 3 – *wnt*, *gp1*, *xf1*, *hm*, etc. – together take 61,200 cycles in the worst case. Since our MIPS R4K core runs at 100 MHz, the rate constraint allows us to use 100,000 cycles. Thus, we have 38,800 cycles left for the remaining tasks – *oh0*, *oh1*, *fk*, *mvm*, *cjd* and *cg*. The *wcet* found in Example 10 fits our rate constraint (note that the two schedules considered in Example 2 would violate the constraint). Thus, our schedule guarantees that we meet our hard real-time rate constraint.

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
context-switch	N/A	42	34	126
priority-sch-sw	107	141	26	44

Table 7: 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	4,500	4,500
cg	2897	59,587	11,000	11,000
run-time-sch-hw	484	413	N/A	99,701

Table 8: Results for the synthesis of hw-tasks.

Table 7 presents the results for the compilation of the software and best- and worst-case execution time estimation with CINDERELLA-M. Unfortunately, due to the presence of interrupts and context switches, we had to turn CINDERELLA-M's instruction cache analysis capabilities off. In Table 8, 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 8 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 addressed the important problem of real-time analysis in hardware-software codesign with a custom run-time system. The CLARA Real-Time Analysis tool, in conjunction with the SERRA 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 static priority assignment, a custom priority scheduler, and a synthesized run-time scheduler, which allows a more detailed analysis of the system. 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.

For our future work we plan to address more fully the issues of semaphores and hardware-software partitioning of the run-time scheduler.

Acknowledgements

ARPA sponsored this research under grant No. DABT63-95-C-0049. We also acknowledge the contributions of Sera Linardi, who ported CINDERELLA to MIPS, and Toshiyuki Sakamoto, who wrote the Verilog hardware-tasks and implemented interrupts in the MIPS R4K model.

References

[1] 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.

[2] This is the same observation made in Malik et. al., ibid., pg. 69.

[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 pre-emptive 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] F. Balarin, K. Petty, A. Sangiovanni-Vincentelli and Pravin Varaiya, "Formal Verification of the PATHO Real-Time Operating System," *Proceedings of the 33rd Conference on Decision and Control, CDC '94*, December 1994.

[9] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno and A. Sangiovanni-Vincentelli, "Formal Verification of Embedded Systems based on CFSM Networks," *Proceedings of the 33nd Design Automation Conference*, June 1996.

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] A. W. Leigh, *Real Time Software For Small Systems* , Sigma Press, Wilmslow, U.K., 1988.

[19] 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.

[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] F. Hillier and G. Lieberman, *Introduction to Operations Research*, 6th edition, McGraw-Hill, Inc., New York, 1995, pp. 424 - 469.

[23] 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 .