Computer Systems Laboratory Technical Report Report No. CSL-TR-97-739

Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems

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

November 22, 1997

Abstract

We present the SERRA *Run-Time Scheduler Synthesis and Analysis Tool which automatically generates a run-time scheduler 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.*

SERRA*'s real-time analysis generates a priority assignment for the software tasks in the mixed hardware-software system. The tasks in hardware and software have precedence constraints, resource constraints, relative timing constraints, and a rate constraint. A heuristic scheduling algorithm assigns the static priorities such that a hard real-time rate constraint can be predictably met.* SERRA *supports the specification of critical regions in software, thus providing the same functionality as semaphores.*

We describe the task control/data-flow extraction, synthesis of the control portion of the run-time scheduler in hardware, real-time analysis and priority scheduler template. We also show how our approach fits into an overall tool flow and target architecture. Finally, we conclude with a sample application of the novel run-time scheduler synthesis and analysis tool to a robotics design example.

1 Introduction

We consider the design of mixed hardware/software systems, such as embedded systems and robots. We aim at providing Computer-Aided Design (CAD) tools that help bring hardware and software design flows closer together in order to allow designers to make tradeoffs between software and hardware and thus more quickly evaluate design alternatives. An indispensable component to a system of cooperating hardware and software is a run-time scheduler.

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 co-design 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, and a CAD tool, called SERRA, to synthesize the necessary hardware and software for the run-time scheduler as well as analyze the performance of the system.

Approaches to *hardware/software co-design* of embedded systems [14] 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 to differentiate approaches 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 ([19] defines granularity in a slightly different way). For example, [20, 18, 5] can be classified as *homogeneous* and *fine-grained* approaches, while [2, 36] are *heterogeneous* and *coarse-grained*. The method of [32] supports *homogeneous* specification in VHDL with both *fine-* and *coarse-grained* partitioning. We take the *heterogeneous* and *coarse-grained* approach in this paper.

There has been much previous work in hardware-software partitioning [18, 36, 14]. However, system designs modeled by heterogeneous specifications are often already partitioned by designers 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 [9, 36].

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 co-design, 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.

Previous approaches to real time analysis have focused on software [28] since the performance analysis of ASICs is considered a well studied problem already. Rate Monotonic Analysis (RMA) [26] and Generalized Rate Monotonic Analysis (GRMA) [35] 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 [3, 4]. 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 [34]. Unfortunately, this approach is usually impractical for hardware/software co-design because it is difficult to handle a situation where the period and computation times are nondeterministic but bounded, since a period of a LCM does not represent all possible situations [37].

Our formulation is similar to [8, 13, 33, 34, 37]. However, in our case we synthesize a custom run-time scheduler in hardware and software for the application [30]. 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 allowing high CPU utilization. In performance- or safety-critical systems (e.g. a mobile robot control system for capture of a satellite in space) our technique can provide precise real-time bounds.

In the approach of this paper, 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 the custom run-time scheduler generated. While the approach of [6, 7] is more general for verification purposes after the priority for each task is assigned, it does not address the issue of assigning the priorities, nor does it address rate-constraint analysis to find a *worse case execution time (WCET)* for the application implemented in hardware and software. Similarly, [5, 13] assume the task priorities are given and does not guarantee their *WCET* calculations are never exceeded.

The rest of the paper is organized as follows. Section 2 explains our design approach and corresponding requirements. Section 3 describes our target architecture for the system and the run-time scheduler. Section 4 discusses how we model our system of hardware and software. Section 5 presents the real-time analysis and priority generation for software tasks. Section 6 presents how we account for worst-case context switch and scheduling cost when running the application, and presents a heuristic for decreasing *WCET* when allowing tasks to be suspended and context switched out. Section 6.6 presents our support for critical regions. Next, Section 7 shows the flow of the SERRA Run-Time Scheduler synthesis and analysis tool. Section 8 gives some experimental results and presents an example from robotics. Finally, Section 9 concludes the paper.

2 Motivation and CAD Requirements

We aim at supporting system-level design with hardware/software tasks custom designed for a target architecture. We refer to the tasks in hardware as *hardware-tasks* and to the tasks in software as *software-tasks*. We assume the existence of mature high-level synthesis tools and software compilers, as well as intellectual property in the form of processor and controller cores. We assume that the system requires both static scheduling, especially in the coordination of hardware-tasks, and dynamic scheduling, given the inexact delay of software and the randomness of the stimuli coming from the environment. 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 into tasks. We assume tasks model system components of significant sizes, and that the system consists of around ten to a hundred tasks. The tasks are assumed to be written either in Verilog or in C. 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: PUMA Arms (Courtesy of the Computer Science Robotics Lab at Stanford)

Example 1 As a motivational example, consider the set of control algorithms of Figure 2. These algorithms calculate torques for the PUMA robot arms shown in Figure 1.

We assume that the controller manages two arms at the same time, and thus any two of the algorithms may be selected in each execution. An execution of the arm controller must complete calculation of new torques for the arms once every millisecond. Since each arm has six degrees of freedom, only six new torque values need to be communicated for each update; thus, the amount of data flow in the system is small. However, the algorithms ("laws" in robotics terminology) need to maintain floating point matrices representing the kinematics and dynamics of the arms, so that the computation would be difficult to represent concisely in, for example, finite-state machines. This control approach is also drastically different than the fuzzy logic adaptive control in [1].

Figure 2 shows three of the ten different algorithms (laws) used with a PUMA arm; `Ohold2 Law`, `Ohold Law`, and `Jhold Law` are top-level tasks which call subtasks in a particular sequence. The *coarse-grained* partitions of Ohold2 Law, Ohold Law, and Jhold Law contains calls to many common subtasks. Some of the subtasks involve hardware components with timing constraints specified on a cycle basis. □

The CAD requirements for co-design of a system such as Example 1 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

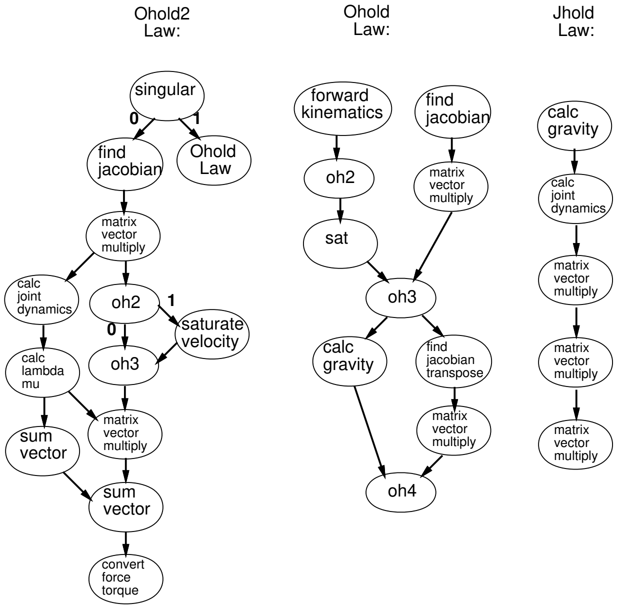


Figure 2: Robotics Example: Concurrent Control Algorithms

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. Third, we need to guarantee a hard real-time rate constraint across tasks in hardware and in software. The handling of multiple-rate constraints is beyond the scope of this paper.

The run-time scheduler synthesis described here supports the execution of software-tasks through an interrupt triggering mechanism where hardware communicates to a software scheduler which of the software-tasks are ready to execute. The CLARA tool, which is embedded within the SERRA system, takes as input the *worst case execution time (WCET)* for each task and then provides for the automated generation of priorities for the software-tasks to be run on a preemptive fixed priority scheduler as well as *WCET* calculation for subsets of hardware- and software-tasks under a hard real-time rate constraint. Thus, SERRA provides the user with the ability to evaluate the performance of different partitions with an automatically generated run-time scheduler (system). For example, the user can migrate a task from C to Verilog to speed up a critical path in the algorithm.

Figure 3 shows the tool flow in which SERRA is embedded. Hardware-tasks are specified in Verilog that can be synthesized by the Synopsys Behavioral Compiler^{TM}[23] (labeled `BC` in Figure 3; `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 a single rate constraint. The implementation of the synthesized system can vary from a system on a chip to a board or set of interconnected components. The overall control/data 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 an RTOS can be added in software by the user if desired, although SERRA's *WCET* calculation assumes that only the software which SERRA generates is run on the microprocessor.

We wrote a new backend for CINDERELLA[28] for MIPS assembly run on a MIPS R4K processor; we call the new tool CINDERELLA-M. Software-tasks are compiled and input to CINDERELLA-M, which outputs a *WCET* for each task. Similarly, from the hardware synthesis of the Synopsys Behavioral Compiler^{TM} (BC^{TM})[23], we obtain an exact execution time for each hardware-task, which we take as a *WCET* for the hardware-task. The *WCET* value for each task is required in order to analyze whether or not we will always meet our rate constraint.

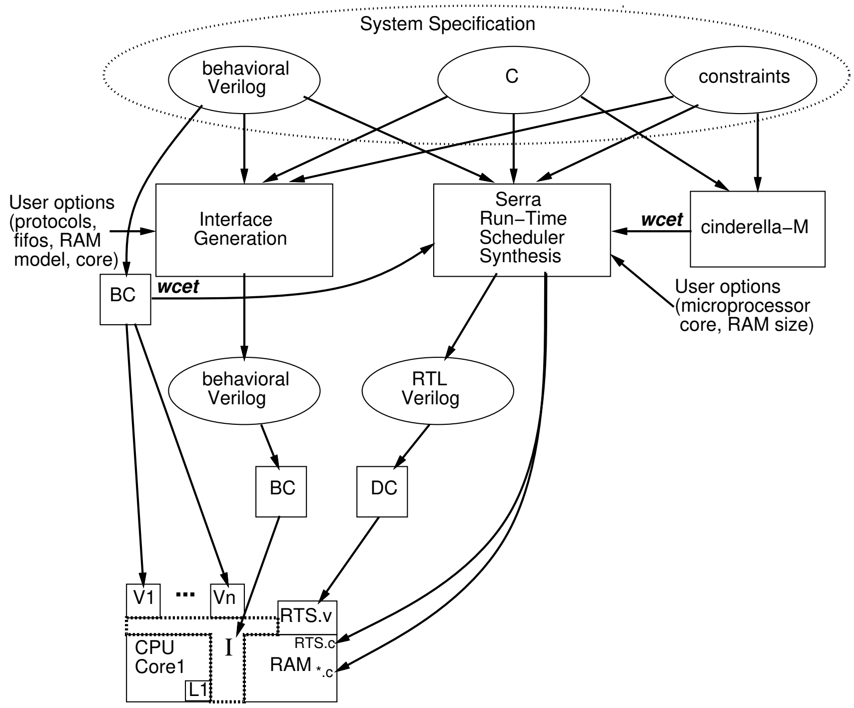


Figure 3: Tool Flow and Target Architecture

This paper focuses on the synthesis and analysis of a custom run-time scheduler.

3 Target Architecture and Kernel

Our target architecture consists of a CPU core with multiple hardware modules, each implementing a particular hardware-task. The CPU has a two-level memory hierarchy consisting of instruction and data caches with a large RAM. Since we target embedded systems, we assume that the RAM is large enough to hold all the program code needed.

3.1 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 2. 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?

The solution to predictability comes from a hardware solution with cycle based semantics. Thus, constraints between events in exact units of cycles can be predictably met. We solve this scheduling problem using a hardware cycle based FSM implementation of the part of the scheduler which chooses which task(s) to execute next.

3.2 Run-Time Scheduler Implementation

We split the run-time scheduler into hardware and software based on an analysis of the constraints. We hypothesize that exact relative timing 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 an RTOS. We try to accommodate both choices by putting in hardware a FSM corresponding to the task control flow 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 preemptive static priority scheduler that executes different threads based on eligible software-tasks as indicated by the *start* vector.

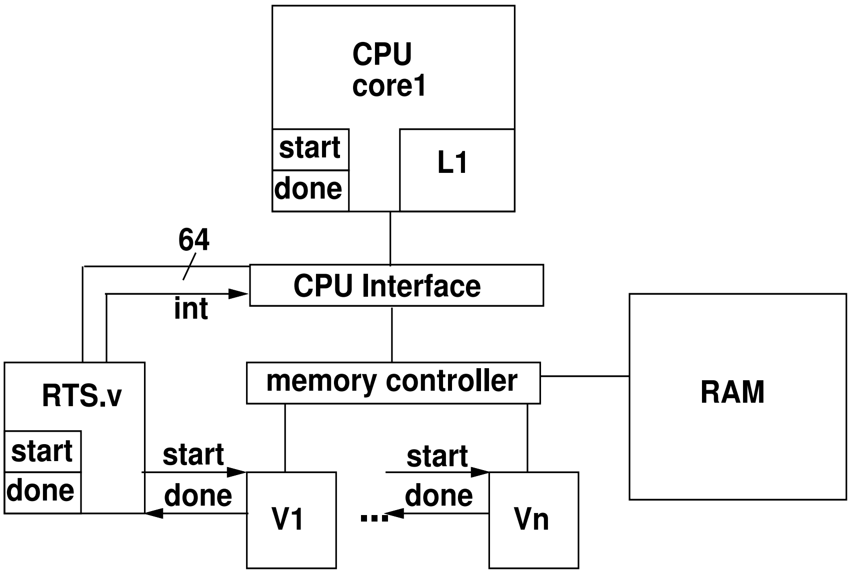


Figure 4: Target Architecture

Figure 4 shows the target architecture of our system. At the top we have a CPU core with a level 1 cache and copies of the start and done vectors in on-chip registers. The bottom shows n hardware tasks $V1$ through Vn. The executive manager hardware FSM is labeled $RTS.v$ and generates all the *start* events as well as receives all the *done* events. This FSM is synthesized to implement 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.

3.3 Control of Software

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* (ISR) reads the register using memory-mapped I/O and places it into the software copy of the *start* vector. Figure 4 shows both the *start* and *done* vectors in registers in $RTS.v$ and their copies in on-chip registers in $CPUcore1$.

The *start* vector may specify that several software tasks are ready to be executed. 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, and whenever a software-task terminates.

When a software-task is finished executing, it updates the *done* vector by writing the new value of *done* 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.

3.4 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. The software tasks are compiled and linked into assembly, with data and program memory statically allocated. Memory-mapped I/O is called with C pointers set explicitly to the appropriate addresses. We thus have a table of software-tasks and their entry points as seen in Table 1.

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

Table 1: Entry Table for Software-Tasks

Therefore, given a particular value of the *start* vector, the appropriate software-task(s) can be executed. The typical sequence of events in software is as follows:

- A hardware interrupt trigger the execution of the ISR.
- The ISR updates the *start* vector and, if a higher priority task has become ready, calls `save_context`.
- A priority scheduler updates the task data structure and executes the highest priority task now ready. If needed, the priority scheduler calls `restore_context`.
- When a software-task is finished, it writes out the new value of the *done* vector.

An advantage of this approach is that it can support standard RTOS scheduling algorithms (round-robin, rate-monotonic, etc.), although we only consider a static priority scheme here. Multiprocessing is helpful when a low-priority, long duration software-task is ready to execute at the same time as a high priority, short duration software-task, but a price is paid when switching context. A disadvantage of multiprocessing is the slower response time due to added overhead for implementing the RTOS scheduling algorithm, polling executive, and associated context switches.

Another possible option which has lower overhead is to have the ISR directly invoke each software-task, executing each task in kernel mode, as discussed in [30]. Such a scheme, however, does not allow a lower priority task to execute while an unexecuted higher priority task is not yet ready. Thus, in this paper we only consider a priority driven scheme.

3.5 Priority Scheduler Template for Software

A task can be in one of two states: *running/suspended* or *ready/terminated*. In our simplified real-time operating system, once a software-task has completed (terminated), it is ready to run again, so we overlap the traditionally distinct *ready* and *terminated* states into one. The *running/suspended* state, combined with the information in the *start* and *done* vectors, tells us whether or not `restore_context` needs to be called before invoking the highest priority task. In particular, if a higher priority task just finished execution and the next highest priority software task ready to execute is in the *running/suspended* state, then we know that it must have been executing earlier at some point. Thus, we execute a `restore_context` for that process. Otherwise, we simply jump to the starting PC for the task.

Note that the interrupt service routine (ISR) is responsible for calling `save_context` if needed. The register file that contains the process state information is saved only when the new *start* vector indicates that a higher priority task is now ready to execute (i.e. we eliminate context switching when one task ends and a new task begins, in which case there is no need to save/restore 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, which will be explained in Section 5. We now turn to modeling issues.

4 System Modeling

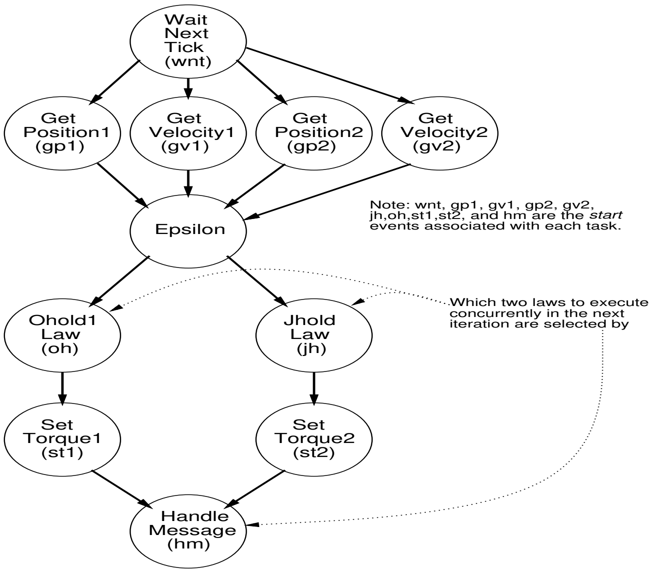


Figure 5: Robotics Example: Main Task

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

Example 2 Figure 5 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 the CDFG of Figure 5 must complete once every millisecond. Thus, we have a rate constraint on the graph.

An example a flattened CDFG where all the nodes are leaf tasks can be seen in Figure 6. The flattened CDFG executes an appropriate subset of the control algorithms of Figure 2 to output torques for two PUMA robot arms. In this case, since there is no branching, the CDFG is equivalent to a DAG with relative timing constraints. □

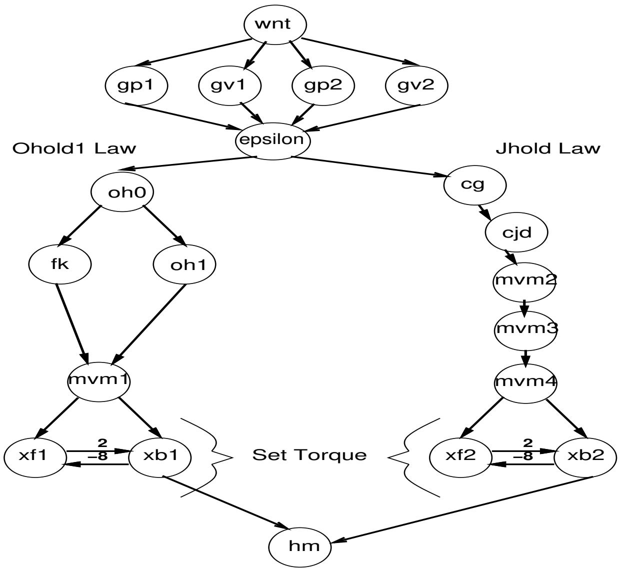


Figure 6: Flattened CDFG of Robot Arm Controller

SERRA leverages previous research on modeling hardware using *control-flow expressions* (CFEs) [10, 11]. In SERRA, CFEs represent an intermediate model of the run-time system that captures the global control-flow information in the system.

CFEs represent the serial/parallel flow of computation, branching, iteration, synchronization and exceptions. CFEs can specify control flow that satisfies our relative timing constraints[24] 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.

We support the specification of tasks that cannot execute concurrently through the use of $NEVER$ sets [10]. For example, $NEVER = \{a, b, c\}$ indicates that tasks a, b, and c can never be active at that same time. In general, $NEVER$ sets can model mutual exclusion; here, we use $NEVER$ sets to model resource constraints. 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 this paper, we consider any number of $NEVER$ sets. For a target architecture of one CPU core, it makes sense to have a single $NEVER$ set of software-tasks, which we use to serialize the software-tasks executed on the same CPU, and multiple $NEVER$ sets of hardware-tasks. This is the case we focus on in this paper.

In a similar vein, 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. Thus, we do not consider $ALWAYS$ sets explicitly in the formulation of our problem.

5 Real Time Analysis

We aim at predictably satisfying real-time constraints in the form of control/data-flow (precedence) constraints, resource constraints, and a rate constraint. We assume that we have as input a CDFG representing the flow of tasks in the application, a rate constraint on the graph, and $NEVER$ sets specifying a resource constraint on software-tasks and resource constraint(s) on hardware-tasks. In this section, we first show a formulation which does not include $NEVER$ sets of hardware-tasks (hardware resource constraints) for the sake of simplicity of explanation. We expand the formulation to include multiple $NEVER$ sets of hardware-tasks in Section 5.2.

To predictably satisfy the rate constraint, we need a *worst case execution time (WCET)* for each task and a $WCET$ for the control/data-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}[23]. We need some assumptions to compute the $WCET$

for the set of tasks.

Assumption 5.1 *We have a Directed Acyclic Graph (DAG) representing a set of tasks,a WCET for each task, and a $NEVER$ set specifying tasks that must be executed in a mutually exclusive manner. A rate constraint is specified for the execution of the whole graph.*

Example 3 Figure 6 shows the DAG resulting from the parallel execution of `Jhold Law` and `Ohold1 Law`. While the full CDFG can select more combinations, e.g. `Ohold2 Law` and `Jhold Law`, we consider here only the case where `Jhold Law` and `Ohold1 Law` are selected to execute in parallel. In other words, the CDFG has been effectively reduced to a DAG. Note that the system is still dynamic since the start and done times of tasks in the DAG are not determined ahead of time but are handled at run-time. Also, the DAG may contain relative timing constraints. □

Note that reducing the CDFG to a DAG limits the amount of control-flow information in the graph to relative timing constraints among tasks. In particular, a control choice equivalent to an `if` statement is not modeled, nor is branching. Also note that for now we consider only a single $NEVER$ set of software-tasks executed on the same CPU. We assume that we have the resulting DAG in graph form $G(V, A)$, where V is the set of vertices and A is the set of directed edges ("Arrows").

Assumption 5.2 *Software is executed by a simple priority scheduler consisting of four code segments: an* `interrupt_service_routine`*(ISR), a* `priority_scheduler`*, a* `save_context` *routine and a* `restore_context` *routine.*

Note that the `priority_scheduler` is compiled for each embedded application; the other three routines are written in assembly and do not require any recompilation.

Assumption 5.3 *Each task, once started, runs to completion.*

Together with the previous assumption and the fact that the `priority_scheduler` code only uses registers reserved for the operating system, we find that the only overhead for software-tasks are the ISR and `priority_scheduler` calls. We will relax the assumption of running to completion later when calculating $WCET$ involving software-tasks which can be partially executed before being interrupted.

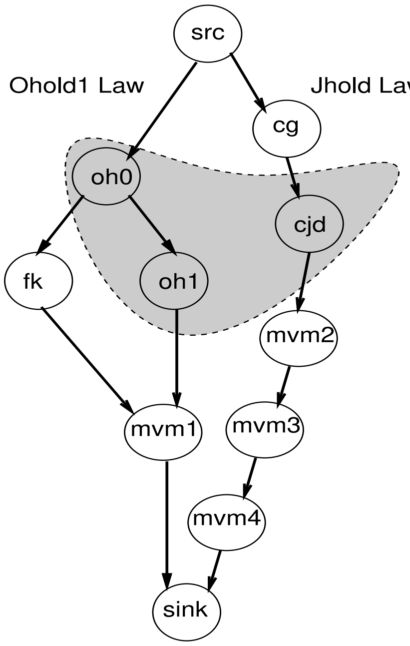
Assumption 5.4 *Hardware-software communication time is included in the WCET of each task and/or is included as a distinct task.*

We have several communication primitives, such as shared memory and FIFOs, with interface generation along the lines of [9, 36].

Assumption 5.5 *Interrupts that switch context come only from the hardware run-time scheduler as described in Section 3.3.*

Example 4 As an example, consider Figure 7. 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`, `forward kinematics`, and `calc gravity`, respectively, in Figure 2. (Task `mvm` has four instantiations in `mvm1-4`.) Similarly, three tasks are specified in C: `oh0`, `oh1`, and `cjd`, where `cjd` corresponds to `calc joint dynamics` in Figure 2 and both `oh0` and `oh1` are coarser-grained groupings of tasks called by `Ohold Law` in Figure 2. 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 7. A first-come-first-serve scheduling algorithm would schedule `oh0` first, then `oh1` (since `mvm` is still executing when `oh0` finishes), and `cjd` last. Without considering the small overhead of the priority scheduler, this results in a $WCET$ of 46,033 cycles for the graph. However, if `oh1` were executed **after** `cjd`, the $WCET$ would be 39,012 for the graph. □



NEVER = {oh0,oh1,cjd}

task	bcet (cycles)	wcet (cycles)
cg	4,000	4,000
oh0	1,598	2,221
oh1	12,341	17,399
fk	4,500	4,500
cjd	9,989	13,213
mvm1	4,400	4,400
mvm2	4,400	4,400
mvm3	4,400	4,400
mvm4	4,400	4,400
src	0	0
sink	0	0

Figure 7: DAG and WCET

Example 4 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 start time of each task in a $NEVER$ set depends upon the scheduling of the other tasks in the $NEVER$ set. In fact, finding the serial order of tasks in the $NEVER$ set which minimizes *WCET* can be shown to be NP-Hard using the Sequencing with Release Times and Deadlines problem [17].

In the context of our system design, solving this problem allows us to proceed with our real-time analysis. For example, once we have a *WCET* for the CDFG of Figure 5, then we can say if the robot controller finishes execution within one millisecond.

5.1 Constructive Heuristic Scheduling

We want to find a schedule for the tasks, with a $NEVER$ set containing all the software-tasks, where the other tasks are all hardware-tasks. We find an ordering of the software-tasks using a problem formulation which is reminiscent of dynamic programming[21]. The formulation enables us to construct in polynomial time a schedule of the tasks which minimizes *WCET* (the heuristic may find a local minimum). Our constructive heuristic scheduling algorithm allows us to take into account precedence constraints, a rate constraint, and a resource constraint in the form of a $NEVER$ set of software-tasks. In Section 5.2, we will extend constructive heuristic scheduling to include multiple resource constraints in the form of $NEVER$ sets of hardware-tasks.

5.1.1 Constructive Heuristic Scheduling Formulation

We take as input the DAG $G(V, A)$ annotated with *WCETs* (one per task), a $NEVER$ set specifying the mutually exclusive software-tasks, $WCETisr$ which is a $WCET$ for the ISR, and $WCETprsched$ which is a $WCET$ for the `priority_scheduler` code.

We divide the problem into stages according to the number of tasks in the $NEVER$ set. We first find a solution for the last stage, then the second-to-last stage, etc., up to the first stage (we proceed in reverse order from the stage number). 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 tasks in the $NEVER$ set plus two (for the source and the sink).

Definition 5.2 *Let t denote a task, and let t_i denote a task executed in stage i.*

Definition 5.3 *Let the multivalued decision variables x_{ik}, $i \in (1, 2, \ldots, n-1)$ and $k \in \mathbf{Z}+$, denote the ordered set of tasks from the $NEVER$ set executed in the subsequent stages, i.e. after stage i.*

Note that x_{ik} represents an ordered set of tasks.

Definition 5.4 *Let X_i, $i \in (1, 2, \ldots, n-1)$, denote the multiset of decision variables $\{x_{ik}\}$.*

Example 5 Consider Figure 7. Since $|NEVER| = 3$, there are 5 stages. In stage 3 we could find that $X_3 = \{x_{31}, x_{32}, x_{33}\} = \{$(oh0,sink),(oh1,sink),(cjd,sink)$\}$. Each x_{3k} is an ordered set, and X_3 is a multiset. □

Definition 5.5 *Let state $s_i = (t_i, x_{ik})$ in stage i denote the current task ready to start execution and the subsequent tasks from the $NEVER$ set executed in stages $(i+1, i+2, \ldots, n-1)$, where the* `sink` *is always executed in stage n.*

Note that given an ordering of software-tasks, the rest of the graph is scheduled with an As Soon As Possible (ASAP) schedule that takes into account the dependencies induced by the ordering of the mutually exclusive tasks.

Example 6 In Figure 7 the 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 $t_4 =$ `oh1` and $t_4 =$ `cjd`. Thus the possible states in stage 4 are $s_4 =$ (`oh1`,`sink`) and $s_4 =$ (`cjd`,`sink`). □

We denote the $WCET$ for task t by $WCET(t)$.

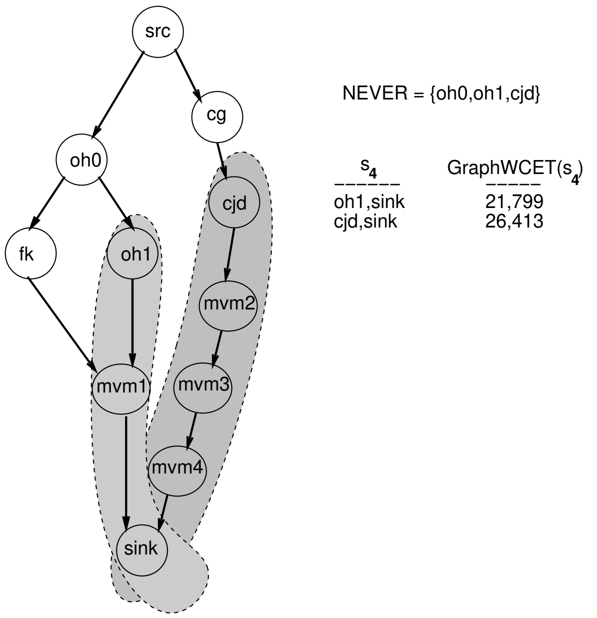


Figure 8: *GraphWCET* Example

Definition 5.6 *Given a state s_i, let $G_{s_i} \subseteq G$ be the directed acyclic graph $G_{s_i}(V_{s_i}, A_{s_i})$ defined by the tasks in state s_i and their successors.*

Example 7 Consider Figure 8. In this example we are in stage $i = 4$. The leftmost shaded area covers G_{s_4} defined by $s_4 =$ (`oh1`,`sink`). In this case $V_{s_4} = \{$`oh1`,`mvm1`,`sink`$\}$. □

Definition 5.7 *Given a state s_i, let s_i be called* **valid** *if G_{s_i} does not contain any task which is in the $NEVER$ set but does not appear in s_i.*

Example 8 Consider Figure 8 again. The two valid states in stage 4 are s_4 = (oh1,sink) and s_4 = (cjd,sink). State s_4 = (oh0,sink), however, is not a valid state because G_{s_4} contains oh1, which is in the $NEVER$ set but not in s_4. □

Definition 5.8 *Given a valid state s_i, let $GraphWCET(s_i)$ be the worst case execution time (WCET) as determined by an As Soon As Possible (ASAP) schedule for G_{s_i}, where any tasks in G_{s_i} which are in the $NEVER$ set are executed in the order in which they appear in s_i. (If s_i is not valid, then $GraphWCET(s_i)$ is undefined.)*

Example 9 Continuing with Figure 8, consider the leftmost shaded area again. For this G_{s_4}, we find that $GraphWCET(s_4) = WCET(\texttt{oh1}) + WCET(\texttt{mvm1}) = 21,799$. □

In other words, $GraphWCET(s_i)$ is the overall *WCET* for stages $(i, i+1, \ldots, n)$, given that the first task t_i in s_i is executed in stage i, and the rest of the tasks x_{ik} in s_i are executed in stages $i+1, i+2, \ldots, n$ according to the order in which the tasks appear in x_{ik}.

Definition 5.9 *Let $f_i(s_i)$, $i \in (1, 2, \ldots, n-1)$, denote a value equal to $GraphWCET(s_i)$ if both s_i is valid and the order of tasks in s_i does not violate any precedence constraints; otherwise let $f_i(s_i) = \infty$. We define $f_n(s_n)$ to be zero since there is no task to execute after the last stage, and the last task executed is always the* sink *(so that it is always the case that s_n = (*sink*)), whose execution takes zero cycles.*

Example 10 A possible state for Figure 8 is $s_4 = (t_4, x_{41})$ = (oh0,sink). However, this state is not valid, and so for s_4 = (oh0,sink), $f_4(s_4) = \infty$. The other two possibilities for s_4 are shown in Figure 8, and for those two we have $f_4(s_4) = GraphWCET(s_4)$. □

Recall that tasks not in the $NEVER$ set are all hardware-tasks and are scheduled ASAP.

Definition 5.10 *Let $f_i{}^*(s_i)$, $i \in (1, 2, \ldots, n-1)$, be the minimum finite value of $f_i(s_i) = f_i(t_i, x_{ik})$ over all possible x_{ik} for a given t_i.*

Definition 5.11 *Given task t_i (the current task executing), let $x_{ik}*$ denote the value of x_{ik} that yields $f_i{}^*(s_i) = f_i{}^*(t_i, x_{ik})$.*

Note that if there is no x_{ik} such that $f_i(s_i) = f_i(t_i, x_{ik})$ is finite, then we have no $f_i{}^*(s_i)$ nor $x_{ik}*$ defined for task sequences beginning with task t_i in this stage.

Thus, when computing $f_i{}^*(s_i)$, we find the following holds, if there exists at least one state x_{ik} for which $f_i(t_i, x_{ik})$ is finite:

$$f_i{}^*(s_i) = \min_{x_{ik}} f_i(t_i, x_{ik}) = f_i(t_i, x_{ik}*), i \in (1, 2, \ldots, n-1)$$

Definition 5.12 *Given a valid state $s_i = (t_i, x_{ik})$, let t_s denote a successor of task t_i, where $G_{t_s} \subset G$ is the graph defined by t_s and the successors of t_s. Then, we define $GraphWCET_{succ}(t_i, x_{ik})$ to be the largest $GraphWCET(t_s, x_{ik})$ of any successor t_s of task t_i.*

In calculating $GraphWCET_{succ}(t_i, x_{ik})$, we schedule the subgraph induced by the successors of task t_i using an ASAP schedule. If we find a successor t_s of t_i that is in the $NEVER$ set, then we use $GraphWCET(t_s)$, which, since the state is valid, was already calculated in a previous stage that scheduled the tasks in x_{ik}.

Example 11 Consider Figure 9 where we are in stage 3; the leftmost shaded area shows G_{s_3} for s_3 = (oh0, oh1,sink). So we have t_3 = oh0 and x_{32} = (oh1,sink). One successor of task t_3 is oh1, which is a member of the $NEVER$ set. Thus we use $GraphWCET(oh1, x_{32}) = 21,799$ as calculated in the previous stage. The other successor of task t_3 is fk, for which we find that the subgraph consisting of tasks {fk, mvm1, sink} yields $GraphWCET(\texttt{fk}, \texttt{sink}) = 8,900$ (recall that a state $s_i = (t, x_{ik})$ consists of a task t and decision variable x_{ik}, where the tasks in x_{ik} must be in the $NEVER$ set, or be the src or the sink, but the task t need not be in the $NEVER$ set). The final result is $GraphWCET_{succ}(t_3, x_{32}) = 21,799$. □

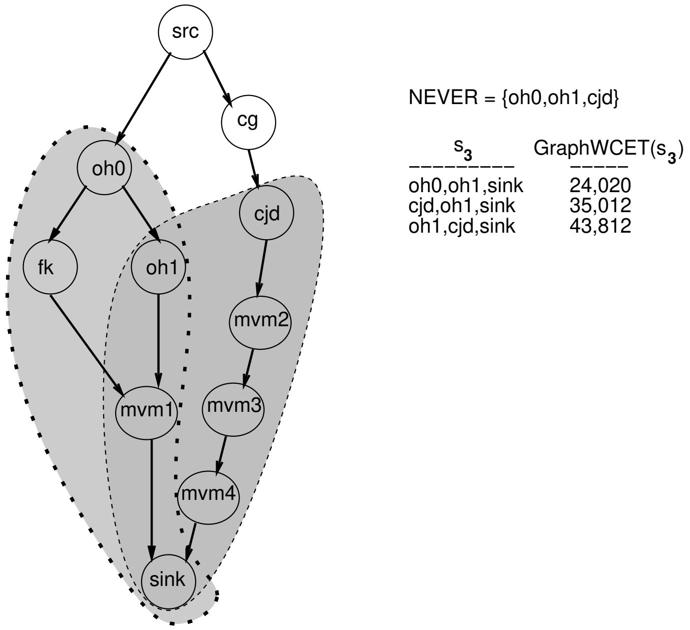


Figure 9: Constructive Heuristic Scheduling Example Stage 3

Definition 5.13 *Given a state $s_i = (t_i, x_{ik})$, we define the following:*

$$GraphWCET_{extra}(s_i) = \begin{cases} GraphWCET_{succ}(t_i, x_{ik}) - GraphWCET(x_{ik}) & \text{if } GraphWCET_{succ}(t_i, x_{ik}) \\ & \quad > GraphWCET(x_{ik}) \\ 0 & \text{otherwise} \end{cases}$$

and $GraphWCET(s_i) = WCET(t_i) + GraphWCET_{extra}(s_i) + GraphWCET(x_{ik})$.

Example 12 Consider the leftmost shaded area in Figure 9 again; we have $t_3 = \texttt{oh0}$, $x_{32} = (\texttt{oh1},\texttt{sink})$ and $s_3 = (t_3, x_{32})$. We found previously that $GraphWCET(x_{32}) = 21,799$ and $GraphWCET_{succ}(t_3, x_{32}) = 21,799$. From these values we find that $GraphWCET_{extra}(s_3) = 0$ and thus $GraphWCET(s_3) = WCET(\texttt{oh0}) + 0 + GraphWCET(x_{32}) = 24,020$. □

This definition allows us to take into account the case where $GraphWCET_{succ}(t_i, x_{ik})$, the $WCET$ of the subgraph covered by the successors of task t_i, is not determined by $GraphWCET(x_{ik})$ (i.e. the path through the software task(s) in x_{ik}) but instead is determined by a different path through the subgraph. At this point we have specified all the definitions needed to calculate $GraphWCET(s)$ for any state s.

5.1.2 Constructive Heuristic Scheduling Solution

The number of stages n we use is equal to the number of tasks in the $NEVER$ set (which we call $SWNEVER$ since it is composed entirely of software-tasks) plus two (for the source and the sink). We use a bottom-up approach and 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 each stage, we compute the best sequence of tasks given that we start with a particular task. That is, in stage i, for each possible first task $t_i \in SWNEVER$, we find the sequence of tasks starting with t_i in stage i and $x_{ik}*$ in stages $(i, i+1, \ldots, n)$ which yields the smallest $GraphWCET$. Thus, since each distinct sequence of tasks defines a unique decision variable for the next stage, at most $|SWNEVER|$ decision variables are carried over from one stage to the next. Thus, for each $t_i \in SWNEVER$, there are at most $|SWNEVER|$ candidates for x_{ik}. This limits the total number of task sequences considered in each stage to a maximum of $|SWNEVER|^2$, making the algorithm polynomial instead of exponential. Unfortunately, it also makes the algorithm a heuristic instead of an exact solution method.

Since the sink is always executed last and takes no time to complete, we assume that this last stage has already been scheduled when we start. Note that in the following we number the stages $(1, 2, \ldots, n)$. The last stage, scheduling the sink, is assumed to be already done. Thus, our approach starts with the second to last stage, stage $n-1$, and progressively works its way back to the first stage, stage 1. Finally, in the following we use index i to refer to current stage.

The pseudo-code for the Constructive Heuristic Scheduling Algorithm is shown in Figures 10 and 11. The algorithm of Figure 10 calculates the worst-case execution time for a given stage, whereas the algorithm of Figure 11 actually implements the constructive heuristic scheduling algorithm and selects the order for the tasks in $SWNEVER$, which is a single $NEVER$ set of software tasks.

$Calc\_WCET\ (G, SWNEVER, i, n, f_{i+1}{}^*, X_i)\{$
    $X_{i-1} = \emptyset;$
    initialize $f_i{}^*$, f_i;
    for $(j = 1; j < (n-1); j++)\{$
        $t_j = j^{th}$ task in $SWNEVER$;
        for (each task order $x_{ik} \in X_i$) $\{$
            $s_i = (t_j,\ x_{ik});$
            if $(order\_not\_possible(G, t_j, x_{ik}))\ \{$ /* if order not possible due to constraints in G */
                $f_i(s_i) = \infty;$
            $\}$ **else** $\{$
                calculate $GraphWCET_{extra}(s_i)$;
                $f_i(s_i) = WCET(t_j) + GraphWCET_{extra}(s_i) + f_{i+1}(x_{ik});$
                /* note that by definition, $f_{i+1}(x_{ik}) = GraphWCET(x_{ik})$ */
            $\}$
        $\}$ /* $f_i(s_i)$ has now been calculated for all possible x_{ik} for this t_j */

        if ($f_i(s_i)$ finite for some $s_i = (t_j,\ x_{ik})$) $\{$ /* if we did not find all $f_i(s_i) = \infty$ in this iteration */
            $x_{ik}* = x_{ik}$ such that $f_i(t_j, x_{ik})$ is minimized;
            $(f_i{}^*(s_i) = f_i(t_j, x_{ik}*);$
            $X_{i-1} = X_{i-1} \cup \{(t_j, x_{ik}*)\};$
        $\}$
    $\}$
    return $(f_i{}^*, X_{i-1});$
$\}$

Figure 10: Calculate $WCET$ Algorithm

$Solve\_order(G, SWNEVER, WCETisr, WCETprsched)\ \{$
    $\mathbf{n} = |SWNEVER| + 2;$ /* number of stages */
    increase $WCET$ for each task in $SWNEVER$ by $WCETisr + WCETprsched$;
    $\mathbf{f}_n{}^*(s_n) = \mathbf{f_n}(\texttt{sink}) = 0;\ X_{n-1} = \{(\texttt{sink})\};$ /* initial values for stage n-1 */
    for $(i = n-1; i > 1; i--)\{$ /* go through the stages in reverse order */
        $(\mathbf{f_i}{}^*, X_{i-1}) = Calc\_WCET(G, SWNEVER, i, n, \mathbf{f_{i+1}}{}^*, X_i);$ /* record $WCET$ and state */
    $\}$ /* when this loop ends we have calculated $\mathbf{f_2}{}^*$ and X_1 */
    $(\mathbf{f_1}{}^*, X_0) = Calc\_WCET(G, \{(\texttt{src})\}, 1, n, \mathbf{f_2}{}^*, X_1);$ /* record state $x_{01}*$ with minimum $WCET$ from `src` */
    $x_{01}* =$ the first (and only) set in X_0; /* X_0 has only one set since the we passed in $\{(\texttt{src})\}$ to $Calc_WCET$ */
    $G_{WCET} = \mathbf{f_1}{}^*(x_{01}*);$ /* annotate G with minimal overall $WCET$ found */
    $G_{task\_order\_list} = x_{01}*;$ /* record the task order found */
$\}$

Figure 11: Constructive Heuristic Scheduling Algorithm

The algorithm of Figure 11 actually implements the core of the constructive heuristic scheduling algorithm.

For stage n, no calculations are necessary since the `sink` always takes zero time to execute.

Scheduling starts with stage $n-1$, for which each task in $SWNEVER$ either can be scheduled then or cannot be scheduled then. For example, if a software-task t_i has a precedence constraint where another software-task must execute **after** t_i, then clearly t_i cannot be scheduled in stage $n-1$ since no software-task can ever be scheduled after stage $n-1$, – the `sink` is always scheduled last (in stage n). Each software-task which can be scheduled to execute in stage $n-1$ without violating any constraints is placed in a one element set and added to X_{n-2} for the next stage.

Then, for stage $n-2$, we calculate a $|SWNEVER| \times |X_{n-2}|$ table where we place in each table entry the $GraphWCET$ for each state determined by a software-task eligible to execute in this stage $(n-2)$ followed by a software-task that can be executed in stage $n-1$ (if the two software-tasks selected cannot execute in the chosen order due to precedence constraints, the table entry records a $GraphWCET$ of ∞). For each task in $SWNEVER$, we record a decision variable (an ordered set, see Definition 5.3) indicating the sequence starting with that task which has the minimal $GraphWCET$. The decision variables are accumulated in X_{n-3} for the next stage $n-3$.

Next, for stage $n-3$, we again calculate a table of size $|SWNEVER| \times |X_{n-3}|$ where we place in each entry the $GraphWCET$ corresponding to an ordered set of three software-tasks. Each ordered set is composed of a task from $SWNEVER$ followed by two software-tasks from an ordered set in X_{n-3}. Since X_{n-3} can contain at most $|SWNEVER|$ sets, we calculate the $GraphWCET$ for up to $|SWNEVER|^2$ combinations of three sw-tasks. For each task t_{n-3} in $SWNEVER$, we select the decision variable $x_{(n-3)k}*$ which minimizes $GraphWCET(t_{n-3}, x_{(n-3)k}*)$ and add ordered set $(t_{n-3}, x_{(n-3)k}*)$ to multiset X_{n-4} for the next stage $n-4$.

Continuing in this way for stages $(n-4, n-5, \ldots, 3, 2)$, we calculate the $GraphWCET$ for each state composed of a task eligible to execute in that stage followed by a particular order of software-tasks in the previous stage, selecting at most $|SWNEVER|$ task orders to pass on to the next stage. Note that as we decrease the stage number by one, we increase the number of tasks in each ordered set $x_{ik} \in X_i$ by one.

Thus, when we reach stage 1, we consider up to $|SWNEVER|$ task orderings of all tasks in $SWNEVER$, where the first task executed is the `src`. From these possibilities we choose the best and find an order of execution for the tasks in the $SWNEVER$ set yielding the smallest $GraphWCET$ among the orders considered. Note that the final list from which the solution is chosen is composed of task orderings chosen based on the optimality of suborderings along the way, i.e. by selecting the x_{ik} that minimize the overall $WCET$ for the graph (the $GraphWCET$). Since choosing local minima may accidentally kick out a subordering which later turns out to be necessary for the global minimum, this formulation is a heuristic. However, it performs in polynomial time.

We next show the application of the algorithm to our example.

In order to begin with the last stage (i.e. stage $n = 5$), we schedule the sink, yielding $f_5{}^*(\texttt{sink}) = 0$.

For stage $n-1 = 4$, the $WCET$ is determined entirely by the current state (whichever task is chosen to execute). Therefore, our table of calculations need only include s_4, $f_4(s_4)$ and X_4.

X_4	$f_4(s_4)$	
t_4	sink	X_3
oh0	∞	
oh1	21,799	(oh1,sink)
cjd	26,413	(cjd,sink)

Table 2: Constructive Heuristic Scheduling Example Stage $n-1 = 4$

Example 13 Consider Figure 7. We have $n = 5$ stages. For stage 5 we found that $f_5{}^*(\texttt{sink}) = 0$. Table 2 shows the calculations for stage 4. From this we achieve one optimization for the next stage already: oh0 cannot be scheduled in this stage due to control/data-flow (precedence) constraints. Thus, the multiset X_3 calculated for the next iteration only has two members.

Figure 8 showed the two sets of tasks scheduled and their $WCET$ paths in this pass of the algorithm. □

For stages $n-2$ through 2, we use the f_{i+1} and X_i values calculated in the previous iteration. Note that for each possible ordered set of tasks, in the worst case $n * (|V| + |A|)$ operations have to be performed in calculating

$GraphWCET(s_i)$, where V denotes the vertices and A denotes the directed edges in the DAG of the task flow.

X_3	$f_3(s_3)$				
t_3	(oh1,sink)	(cjd,sink)	$x_{3k}*$	${f_3}^*(t_3, x_{3k}*)$	X_2
oh0	24,020	∞	(oh1,sink)	24,020	(oh0,oh1,sink)
oh1	∞	43,812	(cjd,sink)	43,812	(oh1,cjd,sink)
cjd	35,012	∞	(oh1,sink)	35,012	(cjd,oh1,sink)

Table 3: Constructive Heuristic Scheduling Example Stage 3

Example 14 Continuing our attempt to schedule Figure 7, we pass now to stage 3. Table 3 shows the calculations for this stage. The first finite-valued entry contains the $GraphWCET$ if `oh0` is scheduled in stage 3 and `oh1` in stage 4 (with 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/data-flow constraints. Note also that there is no column for $x_{3k} = (\texttt{oh0}, \texttt{sink})$ since it was not possible to schedule `oh0` in stage 4.

To calculate the $GraphWCET$ values for s_3, given that we execute task t_3 in this stage (3) and the first task in x_{3k} in the next stage (4), requires scheduling the subgraph covered by task t_3, the tasks in x_{3k}, and all of their successors. We use an ASAP schedule.

Figure 9 showed the states scheduled in this stage and and their $WCET$ paths in this pass of the algorithm. □

X_2	$f_2(s_2)$					
t_2	(oh0,oh1,sink)	(oh1,cjd,sink)	(cjd,oh1,sink)	$x_{2k}*$	${f_2}^*(t_2, x_{2k}*)$	X_1
oh0	∞	46,033	37,233	(cjd,oh1,sink)	37,233	(oh0,cjd,oh1,sink)
oh1	∞	∞	∞			
cjd	37,233	∞	∞	(oh0,oh1,sink)	37,233	(cjd,oh0,oh1,sink)

Table 4: Constructive Heuristic Scheduling Example Stage 2

Example 15 Next consider stage 2 of the attempt to schedule Figure 7 using the constructive heuristic scheduling algorithm. Table 4 shows the calculations for this stage. For the states beginning with task `oh0`, the minimum value of f_2 is selected by $x_{2k}*$ yielding one value for ${f_2}^*$. Note that $x_{2k}*$ is a set that takes on two different values, namely (`cjd`,`oh1`,`sink`) and (`oh0`,`oh1`,`sink`), in the course of the calculation. On the other hand, X_1 is a multiset that contains all of the sets in its column, so $X_1 = \{(\texttt{oh0}, \texttt{cjd}, \texttt{oh1}, \texttt{sink}), (\texttt{cjd}, \texttt{oh0}, \texttt{oh1}, \texttt{sink})\}$. □

Note that the states eliminated in calculating ${f_i}^*(s_i)$ leave us carrying at most $|SWNEVER|$ ordered sets of tasks to the next stage calculation. This means at most $|SWNEVER|^2$ different possible task orderings are considered in each stage, just as we noted earlier. Unfortunately one of the states eliminated in calculating ${f_i}^*(s_i)$, while suboptimal locally, may turn out to be the global optima. The fact that this algorithm is a heuristic can be verified by applying it to the example of Figure 12.

X_1	$f_1(s_1)$				
t_1	(oh0,cjd,oh1,sink)	(cjd,oh0,oh1,sink)	$x_{11}*$	${f_1}^*(t_1, x_{11}*)$	X_0
src	39,012	41,233	(oh0,cjd,oh1,sink)	39,012	(src,oh0,cjd,oh1,sink)

Table 5: Constructive Heuristic Scheduling Example Stage 1

Example 16 Now for the last set of computations, stage 1. There is only one starting state, the source, so the table has only one row. Table 5 shows the calculations for this stage. The minimum $WCET$ for the graph is found in choosing $x_{11}*$. Note that the algorithm finally takes into account the $WCET$ for task cg, making the option of selecting `cjd` to execute before `oh1` less favorable. We end up with $X_0 = x_{01}*$, and so the order found is $x_{01}* =$ (`src`,`oh0`,`cjd`,`oh1`,`sink`) with a $WCET$ of 39,012. Thus we give `oh0` the highest priority, `cjd` the second-highest, and `oh1` the lowest priority. Note that we use X_0 and $x_{01}*$ only to record the final order found (there is no stage 0). □

Thus we have an order (given our assumptions) of execution of tasks in the $NEVER$ set which minimizes $WCET$ from among the task orders considered. We use this order to statically set the priorities for the software-tasks.

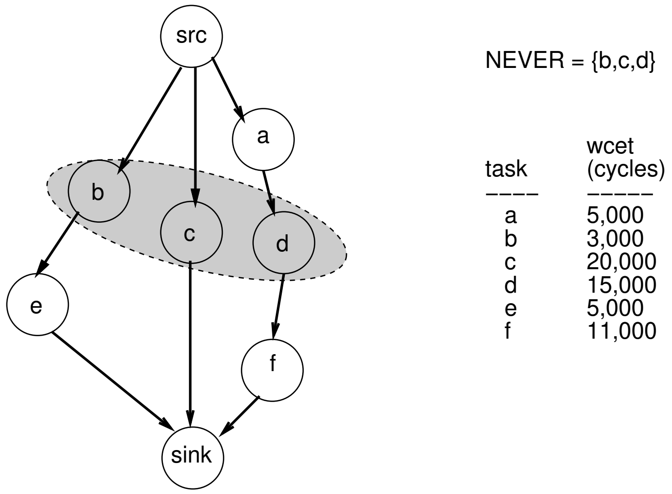


Figure 12: Sample DAG With Optimal Schedule Not Found By Heuristic: The constructive heuristic scheduling algorithm finds order (`d`,`b`,`c`) which yields a $WCET$ of 43,000; however, the optimal order is (`b`,`d`,`c`), which yields a $WCET$ of 40,000.

5.2 Multiple $NEVER$ Sets of Hardware-Tasks

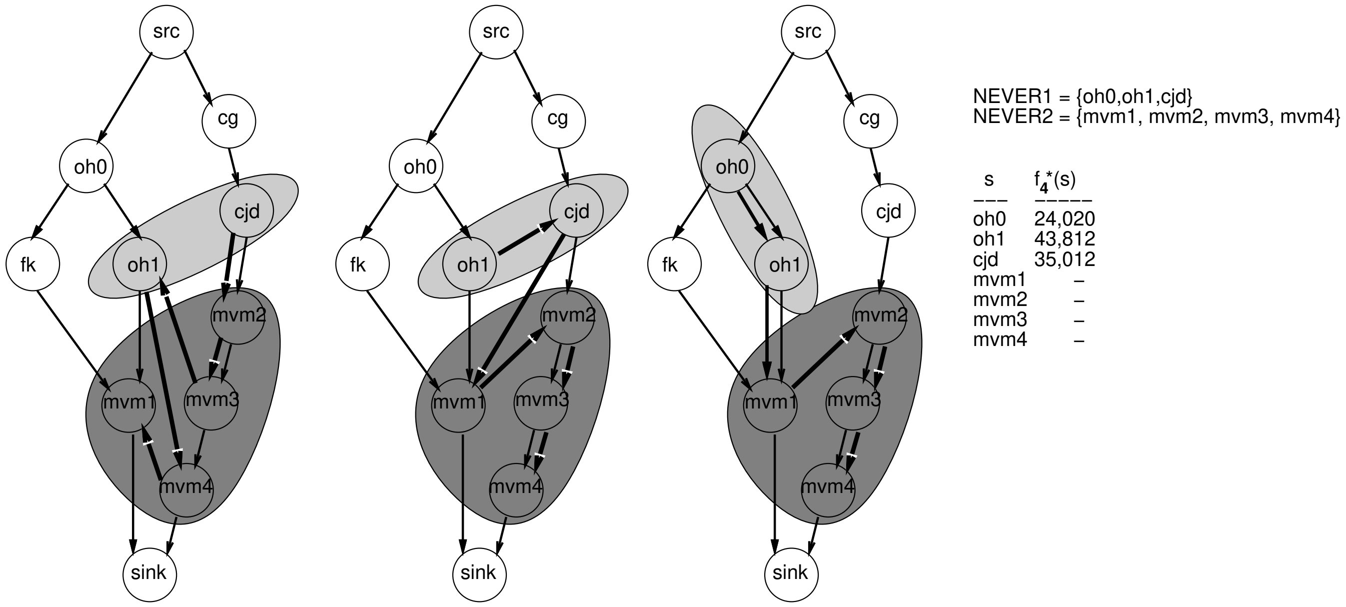


Figure 13: Multiple $NEVER$ Set Example

Up till now we have formulated our scheduling problem under the assumption that we have unlimited hardware and a single processor. Now suppose we do have limited hardware resulting in hardware-tasks implemented on the same hardware resource. We represent each such resource constraint with a $NEVER$ set of mutually exclusive hardware-tasks which cannot overlap execution.

We can include multiple $NEVER$ sets of hardware-tasks by extending the constructive heuristic scheduling algorithm in a straightforward fashion. We simply set the number of stages n equal to the total number of tasks in all $NEVER$ sets, plus two for the `src` and `sink`. Let the number of distinct $NEVER$ sets be d, where the first $NEVER$ set contains all software-tasks in the application, while subsequent $NEVER$ sets contain hardware-tasks which utilize the same hardware resource to accomplish their computation.

Example 17 We consider a modified version of Figure 7 where the four tasks `mvm1-4` are all executed on the same hardware module `mvm`. Figure 13 shows the six of the seven tasks in the two $NEVER$ sets as they are scheduled in stage 4 of the constructive heuristic scheduling algorithm. We have $n = 9$ stages, so $f_9{}^*(s)$, $f_8{}^*(s)$, $f_7{}^*(s)$, $f_6{}^*(s)$ and $f_5{}^*(s)$ have already been calculated. The shading in Figure 13 identifies the tasks in the same $NEVER$ set scheduled at this step of the algorithm; the thick arrows indicate the relative ordering among all of the tasks. The table for this stage is not shown here but would look similar to Table 4. except that it would have seven by seven entries, one row/column per task in a $NEVER$ set.

Note that at this stage we have already scheduled 5 tasks and are considering which task to schedule just before those 5. Due to precedence constraints in the DAG, none of `mvm1-4` can be scheduled at this stage, and therefore the entries are empty. (For example there is no way to schedule `mvm1` in this stage and thus have 5 tasks scheduled **after** `mvm1` completes.) □

The constructive heuristic scheduling algorithm has already been shown in Figures 10 and 11. The only difference in calling the algorithm is that instead of passing in $SWNEVER$, we call it with a multiset $NEVERSETS$ which contains the first $NEVER$ set equal to $SWNEVER$, while the rest of the $NEVER$ sets all contain only hardware-tasks. The major difference from the single $NEVER$ set algorithm shown in Figures 10 and 11 occurs in scheduling the DAG at each step in the algorithm. Instead of a single ASAP schedule for the entire graph, we have to perform an ASAP scheduling of the graph for each distinct never set. Thus, in the worst case, $(|V| + |A|) * d$ operations have to be performed in calculating $GraphWCET_{extra}(s_i)$ of Definition 5.13, where d denotes the number of $NEVER$ sets contained in the multiset $NEVERSETS$.

5.2.1 Complexity Analysis

First note that in order to calculate $f_i(s_i) = f_i(t_i, x_{ik})$, we have to ASAP schedule the DAG $G_{s_i}(V_{s_i}, A_{s_i})$, where V_{s_i} denotes the vertices and A_{s_i} denotes the directed edges (arrows) in the DAG of the task flow of s_i. Note that $G_{s_i}(V_{s_i}, A_{s_i})$ has already scheduled all resource-constrained tasks other than t_i in the previous stage. For each task in $NEVER_i \in NEVERSETS$, an upper bound on the number of constant operations that have to be performed for the ASAP schedule is $((|V_{s_i}| + |A_{s_i}|) * d)$, where d is necessary since each task may have to be visited d times to account for multiple fanins. Since in each stage t_i ranges over the tasks in some $NEVER$ set, and recalling that $n - 2 = \sum_{NEVER_i \in NEVERSETS} |NEVER_i|$, we find that t_i can take on any of $n - 2$ values. Now, since for each possible value of t_i we select at most one value of $x_{ik}*$, X_{i-1} has at most $n - 2$ members in each iteration. Thus, since in each iteration we calculate f_i for every possible state (t_i, x_{ik}), in the worst case $(n-2)^2$ calculations of f_i are needed each iteration. Together with our earlier upper bound of $(|V_{s_i}| + |A_{s_i}|) * d$ for calculating f_i, we end up with an asymptotic upper bound of $O((n-2)^2 * (|V_{s_i}| + |A_{s_i}|) * d) = O((n^2) * (|V| + |A|) * d)$ calculations for one stage (i.e. for $Calc\_WCET$ of Figure 10).

For $Solve\_order$ shown in Figure 11, none of the lines take time greater than $O(n^2 * (|V| + |A|) * d)$. Thus, since we call $Calc\_WCET$ at most n times, our constructive heuristic scheduling algorithm with multiple $NEVER$ sets takes time $O(n^3 * (|V| + |A|) * d)$. Assuming we can bound V, A, and d with a polynomial of n, we have a polynomial-time algorithm.

5.3 Practical Considerations for the Calculation of *WCET*

In order to make a correct calculation of the *WCET*, we have to consider the time spent executing the ISR and the `priority_scheduler`. To be more specific, we will consider the case where the processor is a MIPS R4K. 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 (with no cache analysis, i.e. assuming we always miss in the instruction cache): interrupt overhead = 38 cycles and priority scheduler task selection = 98 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. The priority scheduler template uses several registers reserved for the kernel; it also uses two general purpose registers, which it saves before using and restores just before exiting. Otherwise, with a general context switch, our interrupt overhead would be much larger. Also, since each task runs to completion (Assumption 5.3), no context switches are needed between tasks (in the following sections, we will show how to relax this assumption and still account for the worst case).

We use these costs to calculate the $WCET$ of the entire graph. Note that in the actual implementation of the constructive heuristic scheduling algorithm, the $WCET$ for the ISR and the $WCET$ for the priority scheduler are added to the $WCET$ for each software-task when calculating the task priorities.

We use the priority scheduler with the priorities found via constructive heuristic scheduling. Note that we assume that precedence constraints needed to implement the chosen task order is enforced by the run-time scheduler. In other words, no interrupts updating the *start* vector of *start* events for the software-tasks for a particular software-task until all higher-priority software-tasks are finished executing

Example 18 Consider Figure 7. We use the priorities found in Example 16. We find that the run-time scheduler causes three interrupts. Since the hardware part of the run-time scheduler enforces the precedence constraint of `cjd` before `oh1`, 1,643 clock cycles go unused between the completion of `oh0` and the start of `cjd`. After the third interrupt, `oh1` executes concurrently with `mvm2`, `mvm3` and `mvm4`. After `oh1` finishes, then `mvm1` executes.

A straightforward ASAP schedule is used. Several of the software- and hardware-tasks have loops, for each of which the user provided upper bounds (the analysis of CINDERELLA-M supports user specification of loop bounds[28]). Notice how the critical path runs through both hardware and software in different execution paths. Table 6 shows the calculation. The overall

sw-task	# cycles	hw-task	# cycles
int-ser-routine	38	cg	4,000
priority-sch-sw	98		
oh0	2,221		
int-ser-routine	38	fk	4,500
priority-sch-sw	98		
cjd	13,213		
int-ser-routine	38	mvm2	4,400
priority-sch-sw	98		
oh1	17,399		
”	”	mvm3	4,400
”	”	mvm4	4,400
		mvm1	4,400

Table 6: $WCET$ Calculation Example

$WCET$ is 39,284 cycles. □

Recall that we assume that the hardware part of our run-time scheduler is the only source of interrupts for the CPU (Assumption 5.5). Now we know that we can generate the FSM such that hardware part of the run-time scheduler only interrupts the software to indicate that the next highest priority task is ready to execute once the previous task (in priority level) has completed. Thus, we can guarantee that each software task runs to completion (Assumption 5.3). With these two assumptions, we find that no context switches ever occur in our software (no calls to `save_context` or `restore_context`). Furthermore, only one call to `interrupt_service_routine`(ISR) and one call to `priority_scheduler` are needed per software task. Thus, we find that the final output of our $WCET$ calculation is an upper bound on the $WCET$ of the graph, given the priorities assigned to software-tasks in the same $NEVER$ set. The pseudo code for this $WCET$ calculation was shown in Figure 10.

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 Context Switch Cost and Out-of-order Execution

In the previous section, we found a solution that minimizes $WCET$ when software-tasks are assigned priorities and not executed until all higher priority software-tasks have completed. However, in some cases there may be unused CPU cycles between two software-tasks with consecutive priorities, e.g. if a hardware-task needs to finish to satisfy precedence constraints (captured in the DAG). Thus, we may want to relax Assumption 5.3 and allow lower priority software-tasks to execute during otherwise unused CPU cycles, even when some higher priority tasks have not yet executed. We call this situation *out-of-order execution* because we abandon the exact sequencing of software-tasks according to their priority as was done in the previous section.

However, now our $WCET$ calculation must account for software-tasks which are partially executed and then interrupted. In our analysis the $WCET$ of a context switch is for *either* saving the register set – `save_context` – or for restoring a previous register set – `restore_context`. Since context switching is a major cost to consider when trying to optimize for real time[1], we feel that the savings is worth the effort spent separating the two kinds of contexts switches.

Note that when a particular software-task completes its execution, there are no registers to save when transferring the processor to another software-task. Similarly, when a particular invocation of a software-task first begins execution, there is no register state to load. Eliminating context switches in these cases does not mean that there cannot be other processes switched out; it just means that saving or restoring the register set may not be necessary at that particular instant.

Interrupts are disabled during context switches. The priority scheduler is restarted if an interrupt is received during its execution. Note that due to the construction of the hardware part of the run-time scheduler, at most one interrupt will occur per software-task.

6.1 Upper bound on extra calls to the Priority Scheduler and Context Switch

Suppose we have m software-tasks whose order, assuming each runs to completion, has been found by the constructive heuristic scheduling algorithm described in Section 5. Then, suppose we allow l of the software-tasks to execute out-of-order; that is, for any of the l software-tasks, if it is ready to start before software-tasks higher in priority are ready, we allow it to execute until one of the higher priority tasks is ready to execute. (Clearly, $l < m$ since the highest priority task cannot execute "out-of-order.") Since at most one interrupt will occur per software-task for each execution of the application (as captured in the DAG), the ISR overhead is fixed based on the number of software-tasks. With interleaved execution of software-tasks, however, the number of calls to the scheduler is not fixed. What is the overhead, in terms of extra executions of the priority scheduler and context switch code, incurred by allowing these l tasks to execute early (out-of-order)?

In order to begin our analysis, we define the following:

Definition 6.1 *Let Π assign a priority to each software-task that minimizes $WCET$ if each task runs to completion: if $\Pi(s_a) > \Pi(s_b)$ then the s_a has a higher priority than s_b.*

Presumably we found Π using the constructive heuristic scheduling algorithm of the previous section.

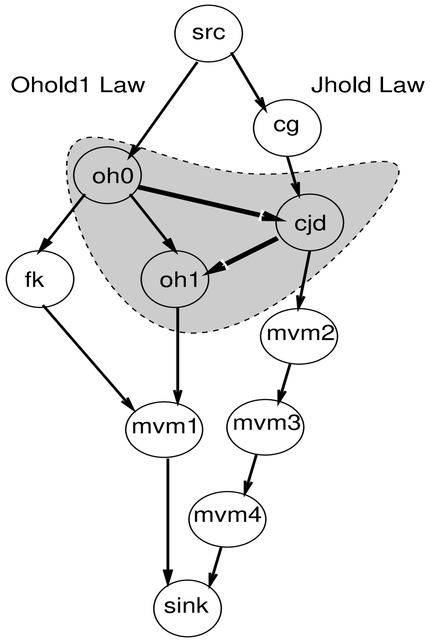
Definition 6.2 *A software-task executes* **early** *when the run-time scheduler sets its start event before all higher priority tasks have completed execution.*

Clearly, a software-task that executes early can possibly execute out-of-order.

Definition 6.3 *Let $I = \{i_1, i_2, \ldots, i_l\}$ = the set of l software-tasks allowed to execute early and possibly execute out-of-order.*

Each task $i \in I$ can have part or all of it computation performed before the software-task immediately preceding it in priority has even begun to execute at all.

[1] For example, the major result of [22] was a 66% reduction in context switch cost.



NEVER = {oh0,oh1,cjd}

task	BCET (cycles)	WCET (cycles)
cg	4,000	4,000
oh0	1,598	2,221
oh1	12,341	17,399
fk	4,500	4,500
cjd	9,989	13,213
mvm1	4,400	4,400
mvm2	4,400	4,400
mvm3	4,400	4,400
mvm4	4,400	4,400
src	0	0
sink	0	0

Π(oh0) > Π(cjd) > Π(oh1)

Figure 14: DAG, WCET and Π Example

Definition 6.4 *Suppose we have two tasks i and j with $\Pi(j) > \Pi(i)$ but under some conditions it is possible that the run-time scheduler will assert the start event for i before the start event for j. Then we say that software-task i can* **jump** *software-task j.*

Clearly, for it to be possible for i to jump j, then there cannot be any precedence constraint between i and j.

Definition 6.5 *Given a set I of software-tasks that can execute early, let $J = \{j_1, j_2, \ldots, j_q\}$ = the set of q software-tasks that can be* **jumped** *by some $i \in I$.*

Example 19 Consider the DAG shown in Figure 14 where the $NEVER$ set specifies software-tasks which must execute on the same CPU. The order of tasks in the $NEVER$ set which minimizes $WCET$ for the graph is (oh0,cjd,oh1) – thus $\Pi(\texttt{oh0}) > \Pi(\texttt{cjd}) > \Pi(\texttt{oh1})$ – and is shown by the two darker arrows in Figure 14. Thus, the static priority scheduler in software has the highest priority assigned to oh0, the next highest priority to cjd and the lowest priority to oh1. Notice that after oh0 finishes, there are 8,779 cycles of delay before cjd can start, due to cg. If the run-time scheduler were to set the start event for oh1 right after oh0 finishes, then oh1 would execute **early** and cjd would be **jumped**. In this case we would have $I = \{i_1\} = \{\texttt{oh1}\}$ and $J = \{j_1\} = \{\texttt{cjd}\}$. □

In general, a task can be in both I and J. Note that in Figure 15, Figure 16, Example 20 and the subsequent proofs, the abbreviation **p** stands for a call to the `priority_scheduler` code, **sc** to the `save_context` code and **rc** to the `restore_context` code.

Example 20 Figure 15 shows a graphical representation of the execution of the DAG of Figure 14 where oh1 executes early (i.e. out-of-order) with respect to its assigned priority (the thick arrows indicate the out-of-order execution flow). The two small columns show which extra calls to the `priority_scheduler` code (**p**), `save_context` code (**sc**) and `restore_context` code (**rc**) occur. An extra call to **p** first occurs to schedule i_1 = oh1 right after oh0 finishes. There is no need to call any context switch code since one software-task is completely finished, namely oh0, and the other software-task, oh1, starts up from the beginning of its code. Next, j_1 = cjd becomes ready, necessitating a call to **sc** to store the register state for oh1. Finally, cjd finishes and a call to **rc** is needed to continue execution of oh1 from its state when it was interrupted. Thus, after the source, i_1 causes an extra call to **p**, j_1 causes an extra call to **sc**, i_1 causes an extra call to **rc** and finally the sink is reached. Thus, the columns show the extra overhead incurred in extra calls to **p**, **sc** and **rc** that would not have been incurred were the tasks executed strictly in order of their assigned priorities. □

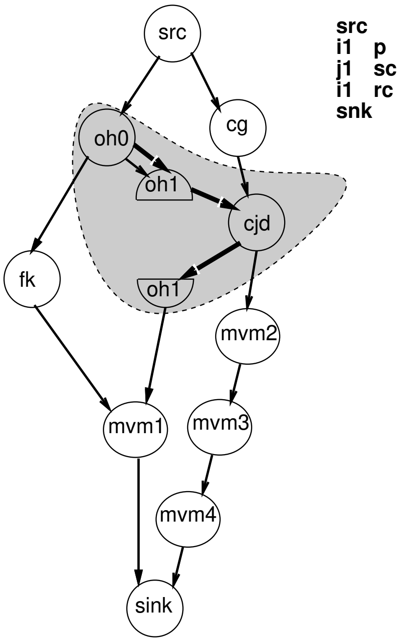


Figure 15: DAG With Out-of-order Execution Example

Example 21 Let's consider the three examples of Figure 16. In **(A)**, $I = \{i_1\}$ and $|I| = 1$; in **(B)**, **(C)**, **(D)** and **(E)**, $I = \{i_1, i_2, i_3\}$ and $|I| = 3$. Notice that in all five examples the number of software-tasks that get "jumped" is $|J| = 10$. Both **(B)** and **(C)** have some tasks in both I and J; for example, in **(B)** i_1 and i_2 can be jumped by i_3, and so both $i_1 \in J$ and $i_2 \in J$.

In **(A)**, i_1 is allowed to execute after the source. So, in every space between two software-tasks, i_1 tries to execute, causing an extra call to **p** and to **rc** before actually running any instructions of i_1 itself. Then, when a task in J is ready to execute, a call to **sc** has to be made since i_1 is not finished yet. Notice that no **rc** calls are needed for any of the tasks in J since each $j \in J$ runs to completion. □

Next we propose two theorems about the number of additional calls to **p**, **rc** and **sc** if we allow software-tasks in a set I to execute while no higher priority tasks are ready (even though some higher priority task has yet to start execution). For the sake of simplicity, note that in the following, given two sets A and B, we use $A - B$ to denote the elements of A not in B.

Theorem 6.1 *Consider o hardware-tasks and m software-tasks $\{s_1, s_2, \ldots, s_m\}$ with priority Π which execute on a single processor.*

Let $I = \{i_1\}$ be a single software-task allowed to execute **early**. *Furthermore, let the software-tasks that i_1 can possibly* **jump** *be $J = \{j_1, j_2, \ldots, j_q\}$, where $\Pi(j_1) > \Pi(j_2) > \ldots > \Pi(j_q)$ and $q < m$.*

Claim: the number of additional calls to the `priority_scheduler`(**p**), `save_context`(**sc**) *and* `restore_context`(**rc**) *code due to allowing the software-task i_1 to execute early has an upper bound of*

$$|J| * (\mathbf{p} + \mathbf{sc} + \mathbf{rc}). \tag{I}$$

Proof. In the worst case i_1 executes before j_1, causing an extra call to **p**, but does not finish execution. Next j_1 becomes ready to execute, causing a call to **p** and **sc**. Since the call to **p** would have happened anyway, only the **sc** call is additional. After j_1 finishes, in the worst case there is exactly enough time for only a single extra call to the **p** and to **rc** for i_1 before j_2 is ready to execute. So, both of these calls occur. Next j_2 is scheduled to execute, but needs an extra call to **sc** to store i_1's register set (since i_1 did not finish). In the worst case, the calls continue in this way until j_q, after which i_1 immediately executes, since it is the next priority task. At this step, only an extra **rc** call is needed for i_1. The total number of extra calls is one **p** for i_1 just before j_1, one **sc** just before executing j_1, then **p** + **rc** + **sc** for j_2 through j_q, and finally one

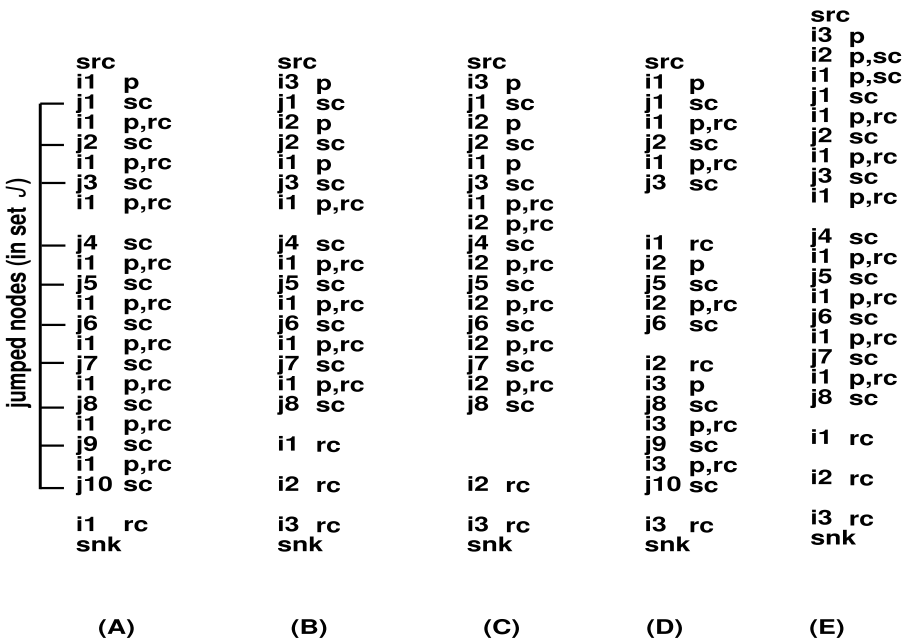


Figure 16: Extra Priority Scheduler and Context Switch Time Examples

rc for the final execution of i_1: $\mathbf{p} + \mathbf{sc} + (q-1) * (\mathbf{p} + \mathbf{rc} + \mathbf{sc}) + \mathbf{rc}$
This is exactly equal to $q * (\mathbf{p} + \mathbf{rc} + \mathbf{sc}) = |J| * (\mathbf{p} + \mathbf{rc} + \mathbf{sc})$. QED. ■

Example 22 An example of the worst case scenario is shown in **(A)** of Figure 16, which shows the case for $q = 10$. The total number of extra calls is $10 * (\mathbf{p} + \mathbf{rc} + \mathbf{sc})$. □

Theorem 6.2 *Consider o hardware-tasks and m software-tasks* $\{s_1, s_2, \ldots, s_m\}$ *with priority* Π *which execute on a single processor.*

Let $I = \{i_1, i_2, \ldots, i_l\}$, *where* $\Pi(i_1) > \Pi(i_2) > \ldots > \Pi(i_l)$, *be software-tasks,* $l < n$, *such that all of them are allowed to execute* **early**. *Furthermore, let the different software-tasks that some* $i \in I$ *can possibly jump be* $J = \{j_1, j_2, \ldots, j_q\}$, *where* $\Pi(j_1) > \Pi(j_2) > \ldots > \Pi(j_q)$.

Claim: the number of additional calls to the `priority_scheduler`(**p**), `save_context`(**sc**) *and* `restore_context`(**rc**) *code due to allowing the software-tasks of I to execute early has an upper bound of*

$$(|(J - (J \cap I)) \cup I| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc}). \tag{II}$$

Proof. We give a proof by induction.

Base step: $I_1 = \{i_1\}$ and $J_1 = \{j_1, j_2, \ldots, j_{q_1}\}$, where J_1 is the set of tasks that i_1 can possibly jump.

Clearly, $I_1 \subseteq I$ and $J_1 \subseteq J$. By Theorem 6.1, an upper bound on the number of calls to **p**, **sc** and **rc** is
$|J_1| * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$.
By definition of J_1 and I_1, $J_1 \cap I_1 = \emptyset$, and so the upper bound is
$|J_1 - (J_1 \cap I_1)| * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$ which, since $|I| = 1$, is equal to
$(|(J_1 - (J_1 \cap I_1)) \cup I_1| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$.
QED for base step.

Step k: $I_k = \{i_1, i_2, \ldots, i_k\}$ and $J_k = \{j_1, j_2, \ldots, j_{q_k}\}$, where J_k are the tasks that some $i \in I_k$ can possibly jump.

Assume true that the following upper bound holds:
$(|(J_k - (J_k \cap I_k)) \cup I_k| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$.
(Note that by definition of J, $J_k \subseteq J$.)

Step $k+1$: $I_{k+1} = \{i_1, i_2, \ldots, i_{k+1}\}$ and $J_{k+1} = \{j_1, j_2, \ldots, j_{q_{k+1}}\}$, where J_{k+1} are the tasks that some $i \in I_{k+1}$ can possibly jump. From the given, we know that $\Pi(i_k) > \Pi(i_{k+1})$. We have several cases.

Case (i): $i_1, \ldots, i_k$ fill all available spaces between tasks in J, so that i_{k+1} is unable to execute out-of-order. By hypothesis (Step k), the upper bound on the number of additional calls to $\mathbf{p}$, $\mathbf{rc}$ and $\mathbf{sc}$ due to J_k and $I_k = \{i_1, \ldots, i_k\}$ is $(|(J_k - (J_k \cap I_k)) \cup I_k| - 1) * (\mathbf{p} + \mathbf{rc} + \mathbf{sc})$ **(1)**. If $J_{k+1} = J_k$, i.e. there are no additional jumpable tasks included in J_{k+1} due to i_{k+1}, then $(|(J_{k+1} - (J_{k+1} \cap I_{k+1})) \cup I_{k+1}| - 1)$ increases by 1 while no additional calls are incurred since i_{k+1} just executes right away, after all the previous tasks in J_{k+1} and I_{k+1} have completed. So the upper bound of $(|(J_{k+1} - (J_{k+1} \cap I_{k+1})) \cup I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$ holds.

So let's assume that there are additional jumpable tasks. Let these additional jumpable tasks included due to i_{k+1} and not already in $J_k \cup I_k$ be $\{j_r, j_{r+1}, \ldots, j_{q_{k+1}}\}$ (we don't consider the jumpable tasks that are also in I because for this case we assume i_k has finished execution).

Just considering tasks $\{j_r, j_{r+1}, \ldots, j_{q_{k+1}}\}$ and i_{k+1}, we have an instance of Theorem 6.1. So the upper bound is $|\{j_r, j_{r+1}, \ldots, j_{q_{k+1}}\}| * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$. If we add this to the previous upper bound **(1)** for $\{j_1, j_2, \ldots, j_{r-1}\}$ and I_k, we have
$(|(J_k - (J_k \cap I_k)) \cup I_k| - 1) * (\mathbf{p} + \mathbf{rc} + \mathbf{sc}) + |\{j_r, j_{r+1}, \ldots, j_{q_{k+1}}\}| * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$ **(2)**.
Now, since $\{j_r, j_{r+1}, \ldots, j_{q_{k+1}}\}$ are not in $J_k \cup I_k$, and since i_{k+1} cannot be in J_{k+1}, we find that
$(J_k - (J_k \cap I_k)) \cup I_k \cup \{j_r, j_{r+1}, \ldots, j_{q_{k+1}}\}$
$= (J_{k+1} - (J_{k+1} \cap I_{k+1})) \cup I_k$
Thus, from **(1)**, we find an upper bound of
$(|(J_{k+1} - (J_{k+1} \cap I_{k+1})) \cup I_k| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$
which is clearly less than
$(|(J_{k+1} - (J_{k+1} \cap I_{k+1})) \cup I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$
QED for case (i).

Case (ii): i_{k+1} was able to execute early, e.g. right after the source but before j_1, because none of $i_1, \ldots, i_k$ were ready to execute or still had execution time left at that time (see Figure 16, **(E)**, for an example). This causes an extra $\mathbf{p}$. However, in the worst case, just as i_{k+1} is about to be dispatched, an interrupt arrives saying that i_k is ready to execute (we do not consider the time due to interrupts here). So, an extra $\mathbf{p}$ and $\mathbf{sc}$ are incurred. Similarly, i_{k-1} becomes ready, incurring yet another $\mathbf{p}$ and $\mathbf{sc}$. This continues for all $i \in I_{k+1}$ in increasing level of priority until we reach i_1. Thus, so far extra calls have occurred in the amount of $\mathbf{p} + (|I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc})$ **(3)**.

(Note that we do not stipulate that all of these interrupts occur before j_1, but only state that in the worst case they arrive in this reverse order and each have enough of a delay before the next interrupt so that additional $\mathbf{p} + (|I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc})$ calls are still made.)

Consider each $j \in (J_{k+1} - (J_{k+1} \cap I_{k+1}))$. From here on out, in the worst case each $j \in (J_{k+1} - (J_{k+1} \cap I_{k+1}))$ will incur an extra call to $\mathbf{sc}$ because j interrupts an executing process. Thus, $|J_{k+1} - (J_{k+1} \cap I_{k+1})| * \mathbf{sc}$ extra calls will occur **(4)**.

Since all tasks in I_{k+1} have become ready to execute, the only way for a task in I_{k+1} to begin execution is if all higher priority $i \in I_{k+1}$ have already finished. In the worst case, i_1 will not finish executing until after the last $j_{q_{k+1}}$ (for the situation where i_1 finishes *before* $j_{q_{k+1}}$, see the next case), so that all of the previous calls to $\mathbf{p}$ for i_1 will have been extra, and only this call to $\mathbf{p}$ now that $j_{q_{k+1}}$ is done will not be an extra call – but the call to $\mathbf{rc}$ will be additional (see Figure 16, **(E)**, for an example with $|I| = 3$). Therefore, $(|J_{k+1} - (J_{k+1} \cap I_{k+1})| - 1) * (\mathbf{p} + \mathbf{rc}) + \mathbf{rc}$ extra calls will be made for i_1 **(5)**.

Similarly, for the rest of I_{k+1}, $|I_{k+1}| - 1$ extra calls to $\mathbf{rc}$ will occur (the remaining $|I_{k+1}| - 1$ calls to $\mathbf{p}$ were necessary in the normal course of events and so are not counted as extra). Thus, a total of $(|I_{k+1}| - 1) * \mathbf{rc}$ extra calls will be needed in the worst case **(6)**.

Combining **(4)**, **(5)**, **(3)**, and **(6)** (in this order), we find that in the worst case the number of extra calls needed is
$|J_{k+1} - (J_{k+1} \cap I_{k+1})| * \mathbf{sc} + (|J_{k+1} - (J_{k+1} \cap I_{k+1})| - 1) * (\mathbf{p} + \mathbf{rc}) + \mathbf{rc} + \mathbf{p} + (|I_{k+1}| - 1) * (\mathbf{p} +$

$\mathbf{sc}) + |I_{k+1}| * \mathbf{rc}$
$= (|J_{k+1} - (J_{k+1} \cap I_{k+1})| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc}) + \mathbf{sc} + \mathbf{rc} + \mathbf{p} + (|I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$
$= (|(J_{k+1} - (J_{k+1} \cap I_{+1}k)) \cup I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$.
QED for case (ii).

Case (iii): Suppose in the previous case i_1 does in fact finish execution before $j_{q_{k+1}}$ (e.g. consider the case in Figure 16, **(C)**, where i_2 executes after j_3 and i_1). In this case an additional call to **p** and and to **rc** are needed for i_2 in the worst case (because i_2 had executed before and needs its context back). However, later on, i_1 will not need to be executed, because it has already finished. This saves calls to **p** and **rc** later: thus, the total amount of calls to **p**, **rc** and **sc** remain unchanged. This can be extended: if any $i_p, p \leq k+1$, finishes early, then i_p will not need to be executed later, leaving the total amount of calls to **p**, **rc** and **sc** unchanged. This is true even if multiple i_p's finish in the same space (i.e. between the same two tasks of set J). Thus the upper bound found in the previous case still holds: $(|(J_{k+1} - (J_{k+1} \cap I_{k+1})) \cup I_{k+1}| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc})$.
QED for case (iii).

Now, since I was chosen arbitrarily, and since J is uniquely determined by I and Π, the above induction holds for any I, Π and corresponding J. QED. ■

The main point of this section has been accomplished: to analyze the worst-case overhead incurred in allowing software-tasks to execute out-of-order. Our major result is Equation II, which gives us a formula which quantifies the number of extra calls to the `priority_scheduler`, `save_context` and `restore_context` code, where we assume that each software-task necessitates one call to the `interrupt_service_routine`(ISR) and `priority_scheduler`.

6.2 Instruction Cache Analysis

We want to quantify all of the overhead associated with allowing software-tasks to execute out-of-order. The previous section dealt with the overhead in terms of extra calls to the priority scheduler and context switch code. What about the instruction cache?

To calculate the *WCET* of a software-task, we use CINDERELLA-M. However, CINDERELLA-M's instruction cache analysis assumes that no interrupts occur [27]. In our case, the presence of interrupts means that a software-task's instructions can possibly be kicked out of the instruction cache if it is suspended to allow execution of a newly ready, higher priority software-task. Thus, we use the following heuristic to augment CINDERELLA-M's analysis.

CINDERELLA-M calculates the binary code size of each software-task. From this, we calculate the maximum number of instruction cache lines neeeded and the cost of reloading the entire intruction cache with the task's instructions. Note that CINDERELLA-M's analysis[28] already includes the worst-case effects for the situation where the binary code size is greater than the instruction cache size, so the maximum number of instruction cache lines we have to consider is bounded by the size of the instruction cache.

Ideally CINDERELLA-M would return a *worst case instruction cache penalty* due to the instruction cache being emptied of a task's instructions. However, we simply read the binary code size using the `View→Function Statistics` command of CINDERELLA-M. Then we use the following formula, where *binarycodesize* and *icachelinesize* are in bytes:

$$WCET\_reload\_icache = (\lceil \frac{binarycodesize - 1}{icachelinesize} \rceil + 1) * (\text{time to load a single icache line}) \qquad \text{(III)}$$

Note the the $binarycodesize - 1$ and $+1$ in the formula are necessary to account for the case where the first instruction byte maps to the last byte of an instruction cache line. The only exception to this formula is when it gives a result greater than the time to load the entire instruction cache, in which case we take *WCET_reload_icache* to be the lower value, i.e. the time to load the entire instruction cache.

Thus, for each possible interruption by a higher priority task that a task can experience, we have to add the cost of reloading all the instruction cache lines for that task to the overall *WCET* for the entire graph. In the worst case, this additional cost will be incurred for every possible call to **rc**. Thus, for each possible call to **rc**, we add the worst case instruction cache refill time, as well as the *WCET* for the `restore_context` code.

Example 23 Software-task `oh1`, when compiled, has a binary code size of 3584 bytes. The `View→Function Statistics` command of CINDERELLA-M is one way to count the binary code size, and this is the method we use. The MIPS R4K we use has icache line size of 16, while the time to load a single instruction cache line is 18 cycles. Thus, for `oh1`, we find the following using Equation III:

$$WCET\_reload\_icache = (\lceil \frac{3584-1}{16} \rceil + 1) * (18) = 4050$$

□

6.2.1 Practical Considerations in Instruction Cache Analysis

As in Section 5.3, we consider the case where the processor is a MIPS R4K. Note that the MIPS R4K does not have a scratchpad section in its primary caches (instruction and data are separate), nor is it configurable to allow one. The cache controller is all in hardware. Thus, in order to calculate the *WCET* of the four operating systems routines we use (ISR, `priority_scheduler`, `save_context`, and `restore_context`), we always assume that they miss in the instruction cache; this assumption was implicit when we calculated these values in Section 5.3.

This estimate is obviously undesirable because the routines are called often and most likely will often be resident in the cache (e.g. none of the three software-tasks considered in our robotics example take up the full instruction cache size of 8K). We could eliminate the instruction cache misses for these four routines in general by either **(i)** finding with other analysis an upper bound on the number of times the routines can be kicked out of the instruction cache, or **(ii)** placing the four routines into a scratchpad section of the instruction cache (i.e. a scratchpad section is one that is never kicked out by the caching system in order to make room for new instructions due to a cache miss). Unfortunately, **(ii)** is not available for the specific CPU we consider.

6.3 Total Upper Bound on *WCET*

In this section we combine the results of the previous two sections in order to come up with a total upper bound formula for the case of tasks with priority Π running on a CPU where the set I of tasks may execute early and the set J of tasks may be jumped.

Let $WCETprsched$ be the *WCET* of the `priority_scheduler` code, $WCETsavecntxt$ be the *WCET* of the `save_context` code and $WCETrestorecntxt$ be the *WCET* of the `restore_context` code. Furthermore, let $WCET\_reload\_icache_i$ be the maximum additional *WCET* due to extra instruction cache misses in task $i \in I$, and let $WCET\_reload\_icache$ be the maximum additional *WCET* due to extra instruction cache misses for any $i \in I$.

Now, for each possible interruption by a higher priority task that a task can experience, we have to add the cost of reloading all the instruction cache lines for that task to the overall *WCET* for the CPU. In the worst case, this additional cost will be incurred for every possible call to **rc**. Thus, for each possible call to **rc**, we add the worst case instruction cache refill time, as well as the *WCET* for the `restore_context` code.

Thus, by Theorem 6.1 and its corresponding Equation I, for a given G and I with one element $\{i_1\}$ and the associated J, an upper bound on the increase in overall *WCET* for G is given by the following:

$$|J| * (WCETprsched + WCETsavecntxt + WCETrestorecntxt + WCET\_reload\_icache) \tag{IV}$$

Similarly, by Theorem 6.2 and Equation II, for a given G and I with associated J, an upper bound on the increase in *WCET* for G is given by the following:

$$(|(J-(J\cap I))\cup I|-1)*(WCETprsched+WCETsavecntxt+WCETrestorecntxt+WCET\_reload\_icache) \tag{V}$$

This completes our calculation of the total upper bound on *WCET* for the CPU with instruction cache analysis included.

6.4 Constructive Heuristic Scheduling with Out-of-order Execution

In this section we present a heuristic algorithm that can improve the solution of the constructive heuristic scheduling algorithm where we do not have Assumption 5.3 and thus software-tasks are not all necessarily atomic.

```
Execute_out_of_order(G, Π, NEVERSETS, WCETprsched, WCETsavecntxt, WCETrestorecntxt) {
    SWNEVER = 1st set in NEVERSETS; m = |SWNEVER|; /* Get set and number of software-tasks */
    Ω = (src, p_1, p_2, ..., p_m) where p_i ∈ SWEVER, 1 ≤ i ≤ m, and Π(p_1) > Π(p_2) > ... > Π(p_m);
                              /* Ω stores the source followed by the software-tasks in priority order */
    p_0 = src;                                        /* now we have Ω = (p_0, p_1, p_2, ..., p_m) */
    W = WCETprsched + WCETsavecntxt + WCETrestorecntxt;
    I = ∅; J = ∅; Ψ = ∅;                               /* Ψ keeps track of new precedence constraints */
    i = 1; WCET_reload_icache = 0;              /* i keeps count of the number of tasks in set I plus one */
      /* in the following for loop, we consider allowing software-tasks (p_m, p_{m-1}, ..., p_2) to execute early */
    for (l = m; l ≥ 2; l − −) {
      k2 = p_{l-1};
      if (k2 ∈ J) v = 0; else v = 1; /* v keeps count of the number of tasks skipped by p_l and not already ∈ J */
      new_prec_task = k2;
      for (k1 = p_{l-2} to k1 = p_0) {
        if (∃ a precedence constraint {k2 → p_l}) continue;              /* exit inner for(k1=...) loop */
        if (get_space(k1, k2) ≥ (i + v + num_tasks_skipped(Ω) - 1)*(W + WCET_reload_icache_i) ) {
          new_prec_task = k1;
          after_prec_task = k2;
          if (WCET_reload_icache_i > WCET_reload_icache) WCET_reload_icache = WCET_reload_icache_i;
        }
        k2 = k1;
        if (k2 ¬∈ J) v + +;
      }
      if (new_prec_task) {
        Ψ = Ψ ∪ {new_prec_task → p_l};   /* add new precedence constraint {new_prec_task → p_l} to Ψ */
        update I, J;
        reduce get_space(new_prec_task, after_prec_task)
          by (i + num_tasks_skipped(Ω))*(W + WCET_reload_icache_i);
        i + +;
      } else Ψ = Ψ ∪ {p_{l-1} → p_l};                       /* add new precedence constraint to Ψ */
    }
    return(Ψ, WCET_reload_icache);
}
```

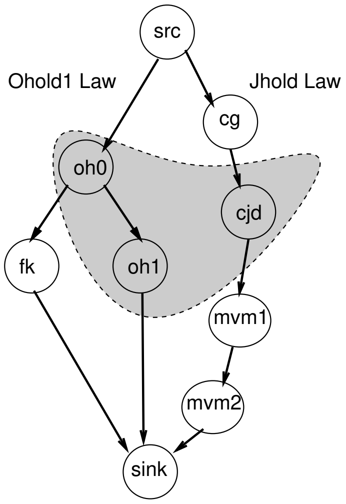
Figure 17: *Execute Out-of-order* Algorithm

We first compute the priorities by the algorithm of Section 5 for multiple $NEVER$ sets. Thus, we have an order of software- and hardware-tasks contained in $NEVERSETS$ and the corresponding $WCET$ for the DAG representing the application. Our goal is to increase CPU utilization by starting execution of a low priority software-task that is ready when no higher priority software-task is yet ready. However, if not done carefully, we could end up increasing overall *WCET*, although in general relaxing Assumption 5.3 will allow us to reduce *WCET* for the graph, thus improving our solution. We use the bounds proven in the previous section to guide our decision and guarantee that any out-of-order execution allowed will not worsen the *WCET*.

The basic insight that we gain from the previous section is the following. Suppose we consider a software-task p_i lower in priority (and thus later in execution if all software-tasks execute strictly in priority order) than two consecutive priority software-tasks $k1$ and $k2$ which leave the CPU unused for a certain number of cycles between the completion of $k1$ and the beginning of $k2$. Let's define a function `get_space`$(k1, k2)$ that returns a number equal to the amount of unused CPU cycles. Should we allow p_i to execute after $k1$ finishes (assuming there are no control/data-flow constraints preventing p_i from doing so)? To answer this question, we use the bound found in Equation V of the previous section: if the amount of space (unused cycles) is greater than or equal to Equation V, then yes, otherwise no. That is the insight behind the heuristic *Execute Out-of-order* procedure of Figure 17.

We describe now the heuristic algorithm of Figure 17 that improves the execution time of a schedule by allowing out-of-order execution. From Π, which was computed by the algorithm described in Section 5.2, we obtain the software-task order $(p_1, p_2, \ldots, p_m)$, where there are m software-tasks. Then we consider allowing a software-task p_l to execute early one at a time in reverse order of the software-tasks from this set (except for the first software-task, for which it does not make sense to execute early). Thus, given a software-task $p_l \in (p_2, p_3, \ldots, p_m)$, and starting with the software-task scheduled last (i.e. p_m), we consider allowing p_l to execute early. For each such software-task p_l we check if p_l can execute in some unused space between two consecutive and higher priority software-tasks $k1$ and $k2$, assuming no precedence constraints are violated. If p_l can execute in the space, then we check if the space is big enough to account for the worst-case extra execution time that will be incurred according to Equation V. Note that we calculate Equation V in Figure 17 by using num_tasks_skipped(Ω), a function which returns the number of tasks currently in $(J-(J \cap I))$ (i.e. not including the tasks currently under consideration, unless they were already placed in I or J in a previous iteration). Now, if the space of unused CPU time is big enough, then we greedily schedule p_l in that space and appropriately reduce the available space to reflect the new schedule. As we go along, we keep track of I and J as we add tasks to each set. Continuing in this way, we consider all possible software-tasks one by one for early execution.

When this heuristic completes, we have a final set of precedence constraints for the software-tasks that allows out-of-order execution without increasing the $WCET$ of the application.



SWNEVER = {oh0,oh1,cjd}

task	task bcet (cycles)	task wcet (cycles)	icache refill wcet (cycles)
cg	11,000	11,000	
oh0	1,598	2,221	612
oh1	12,341	17,399	4050
fk	4,500	4,500	
cjd	9,989	13,213	3258
mvm1	5,000	5,000	
mvm2	5,000	5,000	
src	0	0	
sink	0	0	

Π(oh0) > Π(cjd) > Π(oh1)

Figure 18: Example With $WCET$ Calculation of Instruction Cache Refill Time

Example 24 [Sample Application of *Execute Out-of-order* **algorithm]** Consider Figure 18, which shows the $BCET$ and $WCET$ for each task, the icache refill $WCET$ for the software-tasks ($SWNEVER$), and the priorities found for the tasks: $\Pi(\texttt{oh0}) > \Pi(\texttt{cjd}) > \Pi(\texttt{oh1})$.

We begin by considering oh1 for out-of-order execution. We find that the space between the end of oh0 and the beginning of cjd is 11,000 - (2,221 + $WCETisr + WCETprsched$) = 8,643 cycles (using the costs of Section 5.3, from which we also find that $W = 422$). At this point in the algorithm of Figure 17, we find that $(i + m + \texttt{num\_tasks\_skipped}(\Gamma) - 1) = 1 + 1 + 0 - 1 = 1$, and that $WCET\_reload\_icache_i$ = 4,050, giving us a move cost of $1 * (422 + 4,050) = 4,472$. Since $8,643 \geq 4,472$, we set new_prec_task (of Figure 17) to oh0. We next find out that oh1 cannot execute before oh0 since oh1 requires data generated by oh0. Thus, we add the precedence constraint $\{\texttt{oh0} \rightarrow \texttt{oh1}\}$ which means that we do not add precedence constraint $\{\texttt{cjd} \rightarrow \texttt{oh1}\}$. Therefore, the run-time scheduler will set the $start$ event of oh1 as soon as oh0 finishes execution instead of waiting for cjd to finish.

We next consider cjd for out-of-order execution. We find that it does not make sense to try to have cjd execute before oh0 since oh0 starts right away. So we add the precedence constraint $\{\texttt{oh0} \rightarrow \texttt{cjd}\}$ which means that cjd} will run to completion (since it cannot start until the task immediately preceding it in priority executes). This completes the algorithm of Figure 17 for the example of Figure 18.

Note that the precedence constraint $\{\texttt{oh0} \rightarrow \texttt{oh1}\}$ in this case is redundant because the precedence constraint is already enforced by a control/data-flow constraint (in general, of course, such redundancy will not always be the case).

The result is that the lower priority task oh1 executes in the idle CPU time between the end of oh0 and the beginning of cjd. □

6.4.1 Calculation of *WCET* With Out-of-order Execution

In order to make a correct calculation of the *WCET*, we have to consider the time spent executing the ISR, the priority scheduler, and context switches. As in Section 5.3, we will consider the specific case where the processor is a MIPS R4K. We use the following costs, obtained by analyzing our run-time scheduler software code executed on a MIPS R4K model (with no cache analysis, i.e. assuming we always miss in the instruction and data caches): save context = 162 cycles, restore context = 162 cycles, interrupt overhead = 38 cycles, and priority scheduler task selection = 98 cycles.

After execution of the *Execute Out-of-order* algorithm of Figure 17, we have a maximum value for $WCET\_reload\_icache$ (which could be zero if $|I| = 0$).

With these costs, we calculate the *WCET* of the entire graph, scheduling everything ASAP where each software-task has $WCETisr + WCETprsched = 136$ cycles added to its *WCET*. At this point we have performed exactly the same calculations as in Section 5.3. If $|I| = 0$, then this is our final answer.

If $|I| \neq 0$, then both I and J are nonempty, and we have to account for extra overhead. We use the bound found using Theorem 6.2 in Section 6.3, namely Equation V, reprinted here for convenience:
$(|(J - (J \cap I)) \cup I| - 1) * (WCETprsched + WCETsavecntxt + WCETrestorecntxt + WCET\_reload\_icache)$.
Adding this value to the *WCET* found from scheduling the graph gives us an upper bound on the *WCET* of the graph. This is the value we return to the user.

sw-task	# cycles	hw-task	# cycles
int-ser-routine	38	cg	11,000
priority-sch-sw	98		
oh0	2,221		
int-ser-routine	38		
priority-sch-sw	98		
oh1	8,507		
int-ser-routine	38	fk	4,500
save_context	162		
priority-sch-sw	98		
cjd	13,213		
priority-sch-sw	98	mvm2	4,400
restore_context	162		
WCET_reload_icache	4,050		
oh1	8,892	mvm3	4,400
"	"	mvm4	4,400
		mvm1	4,400

Table 7: *WCET* Calculation Example

Example 25 [WCET calculation] Consider Figure 18. If we make each software-task run to completion, then with the optimal order of (oh0, cjd, oh1) we calculate that the *WCET* for the graph is 46,284 cycles. However, we found in Example 24 that we should allow oh1 to execute after oh0, even though oh1 has a lower priority than software-task cjd. This allows previously unused CPU cycles to be filled.

We have $J = \{\texttt{cjd}\}$, $I = \{\texttt{oh1}\}$ and $J \cap I = \emptyset$. The heuristic of Figure 17 gives us $WCET\_reload\_icache = 4050$. Using our costs for $WCETprsched, WCETsavecntxt, WCETrestorecntxt$ and $WCET\_reload\_icache$, we find that $W = 422$. From Equation V, we find that
$(|(J - (J \cap I)) \cup I| - 1) * (WCETprsched + WCETsavecntxt + WCETrestorecntxt + WCET\_reload\_icache)$
$= 1 * (422 + 4050) = 4472.$
Table 7 shows the ASAP graph schedule with the worst-case execution time added in. Notice that the maximum context

switch overhead and the maximum one additional call to the priority scheduler has been accounted for. The final $WCET$ is 42,113 cycles, which is less than our initial solution of 46,284 cycles. □

This final output is an upper bound on the $WCET$ of the graph given the priorities assigned to software-tasks and the precedence constraints added to the graph and therefore implemented in the hardware portion of the run-time scheduler. In addition to helping to limit the increase in overall $WCET$ due to software-tasks, the added precedence constraints also guarantee mutually exclusive invocation of hardware-tasks in the same $NEVER$ set.

Notice that with this result we do not know exactly *when* each software-task will begin and end. Software schedulers are by their very nature dynamic, especially with a system like ours that contains caches. Thus, a run-time system that statically schedules all software-tasks and their start/finish times may require timers and other additional components, making such an approach infeasible or impractical. Also, the total $WCET$ found for the system may be one that no single static schedule could achieve, because the possibilities for different interactions between tasks could not be so tightly arranged as with the dynamic approach here.

So we now can analyze satisfiability of a rate constraint in a dynamically changing, concurrent execution of hardware-tasks and software-tasks with multiple resource constraints (expressed with $NEVER$ sets), given our run-time scheduler implementation.

6.5 Task Splitting

One of the limitations of the *Execute Out-of-order* algorithm of the previous section is that the original priorities assigned to software-tasks is kept. However, having abandoned Assumption 5.3, one might be tempted to go back to the original formulation of the Constructive Heuristic Scheduling Algorithm of Section 5.1 used to assign priorities. Can we improve upon the algorithm when software-tasks are allowed to execute out-of-order? Are there optimal task priorities with out-of-order task execution which **any** algorithm will always miss because of Assumption 5.3? It turns out that there are. Consider the following example:

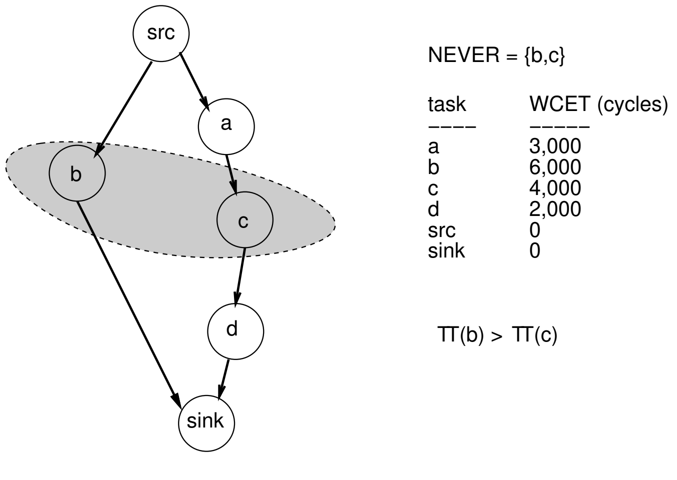


Figure 19: Constructive Heuristic Scheduling Example of Suboptimal Result

Example 26 Consider Figure 19. The constructive heuristic scheduling algorithm will compare the two possible orderings, (b, c) and (c, b), and will find that the overall $WCET$ is 12,000 cycles for the first case and 13,000 for the second. Thus, software-task b will receive the highest priority. Even an exhaustive algorithm which enumerates all possibilities will find this result.

Now we run the heuristic of Section 6.4 and find that we cannot improve on the solution since there is no space (unused CPU cycles) before b, which begins execution right away. Thus c must wait until b finishes to begin execution; overall $WCET$ for the graph is still 12,000 cycles.

Suppose c had a higher priority than b and that out-of-order execution were allowed. Then, ignoring the software scheduling, interrupt and context switch overhead, b would execute for 3,000 cycles concurrently with a, then c would execute for 4,000 cycles, and finally b would finish in 3,000 cycles while d concurrently executes, resulting in an overall $WCET$ of 10,000 cycles, which is significantly less than previously found. □

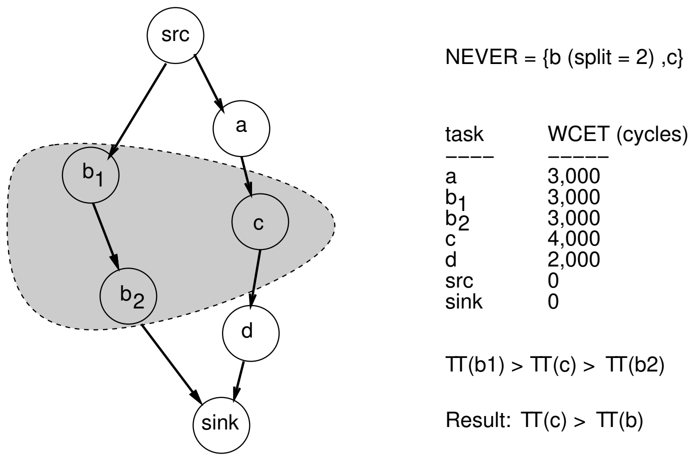


Figure 20: Example of Scheduling with Task Splitting

To deal with this problem, we add the following heuristic: we allow the user to specify for a task s that it can be *split* into n equal chunks. We then split s into n sequential tasks $(s_1, s_2, \ldots, s_n)$ each with $\frac{1}{n}$ of the $WCET$ of s. Then we run the constructive heuristic scheduling algorithm as before, but from the final order we set the priority of s to be the priority found for s_n and discard the priorities found for $(s_1, s_2, \ldots, s_{n-1})$.

Example 27 Consider Figure 20. This time the user specifies that software-task b can be *split* into $n = 2$ chunks. The modified specification of the $NEVER$ set, $WCET$ for each task, and resultant graph can be seen in Figure 20. The constructive heuristic scheduling algorithm finds the ordering (b_1, c, b_2) (which is optimal), from which we extract the order only including b_n, resulting in (c, b_2). Thus c receives a higher priority than b and we have $\Pi(c) > \Pi(b)$.

Now we run the heuristic of Section 6.4 and find that b should be allowed to begin execution right after the source, and then be suspended when c becomes ready. The hardware portion of the run-time scheduler is synthesized to implement this, namely by interrupting the CPU right away to communicate a *start* vector indicating that b is ready to execute. Ignoring the software scheduling, interrupt and context switch overhead, the overall $WCET$ is now 10,000 cycles. □

6.6 Critical Regions

An important programming methodology to support is the use of critical regions. A critical region is a section of software code where critical resource(s) are used or common variable(s) are read/written. In fact, software semaphores were originally created in order to allow the specification of critical regions in software. Thus, if our run-time scheduler can support the specification of critical regions, then we can accomplish the same goal without resorting to semaphores.

We support critical regions via *noninterruptible* software-tasks. The user can specify a set $NONINT$ of noninterruptible software-tasks. If a task is in $NONINT$ then the task will not be considered for membership in the set I of tasks allowed to execute out-of-order. In other words, all higher priority software-tasks must finish **before** the task is scheduled, so that any interrupts received during the task's execution cannot be from a higher priority task, thereby ensuring that the noninterruptible software-task is never kicked out. In this manner a set of critical regions, e.g. that access the same shared variables or other resource, can be defined. The algorithms of Section 5 are modified to take into account that these processes are noninterruptible by simply retaining Assumption 5.3, namely that the task, once started, runs to completion. Note that one could implement a semaphore S by specifying each access to S as a noninterruptible software-tasks.

Since the entire critical region must run to completion, releasing the resource or no longer accessing the shared variable, the *priority inversion* problem does not arise in the final implementation. The problem of

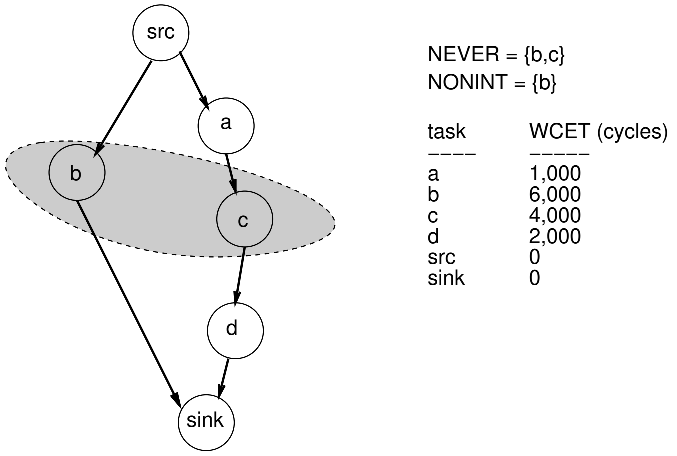


Figure 21: Example Specification of Noninterruptible Task

priority inversion refers to a situation where a lower priority process holds a resource when a higher priority process interrupts which needs to use the held resource. In this case the higher priority process is prevented from executing and has to release control to the lower priority process; thus, the lower priority process has, in effect, made itself higher in priority – i.e., the priorities of the two processes have been inverted. By design, a noninterruptible software-task cannot give rise to the *priority inversion* problem.

Note, however, that we assume that the critical region is located as a single task within a DAG. Thus, the only looping on the critical region or semaphore allowed is that of each execution of the DAG; finer level looping on a critical region is not allowed (although loops without critical regions *are* allowed in individual C and Verilog tasks, as long as an upper bound can be given on the number of times a loop will repeat in a given execution of the task containing the loop).

Example 28 In Figure 21 task b is specified as noninterruptible. CLARA finds that the order, if each task runs to completion, is (c, b). Since b is noninterruptible, we do not consider executing part of b during the unused CPU time available while a is executing. The precedence constraint $\{\mathsf{c} \rightarrow \mathsf{b}\}$ is generated. □

7 Tool Flow

Figure 22 (repeated from Figure 3 for the reader's convenience) shows our tool flow when applying our design tools to a system design. The hardware-tasks are written in Verilog and software-tasks are written in C. Constraints include relative timing constraints, a single rate constraint, and resource constraints in the form of $NEVER$ sets. Precedence constraints are implicit in the task specification which takes the form of a Directed Acyclic Graph. SERRA focuses on run-time scheduler synthesis and *worst-case execution time (WCET)* analysis both to help optimize the scheduler synthesized and to satisfy a rate constraint. Satisfaction of relative timing constraints (minimum and maximum separation) in hardware blocks is dealt with in hardware control synthesized by a tool (THALIA2) not described in this paper [10, 11].

The system-level tasks, written Verilog and C, and the 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 takes input in C and outputs a $WCET$ for each software-task (note that bounds on loops must be provided by the user)[27, 28]. Similarly, from BC^{TM} we obtain a $WCET$ for each hardware-task (loop bounds must be provided here in some cases as well). Since we compare BC^{TM}-generated $WCETs$ with software $WCETs$, we convert all delays to the number of microprocessor clock cycles (since the hardware clock speed is typically slower.) These values are used to annotate the leaf tasks in the final DAG of the system specification. Figure 7 showed a sample DAG and a corresponding table with the $WCET$ annotations.

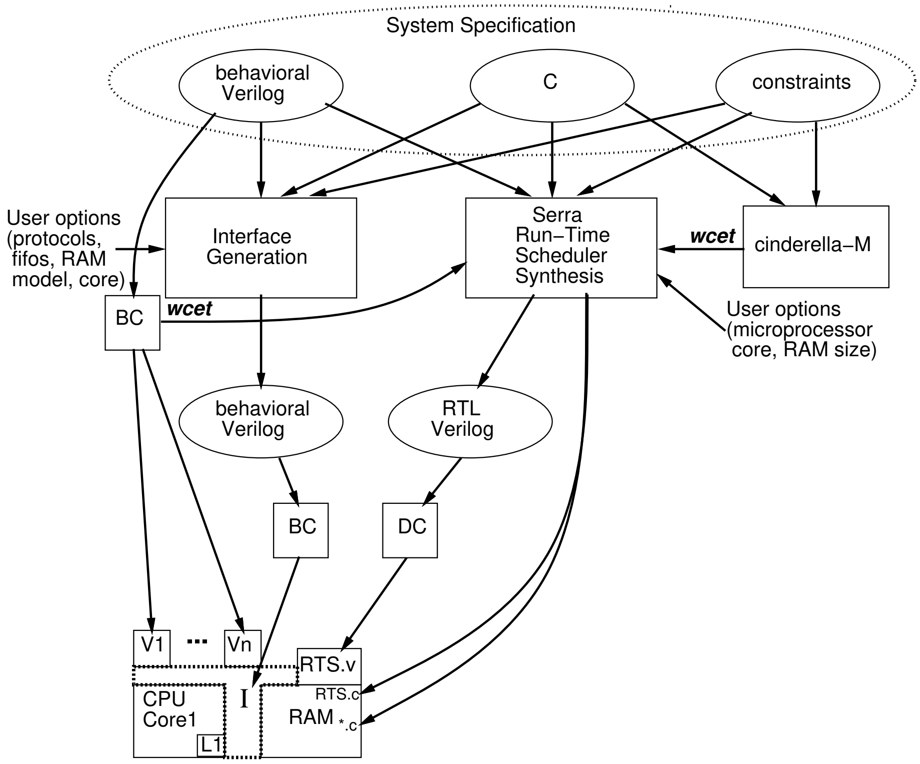


Figure 22: Tool Flow and Target Architecture

7.1 SERRA Run-Time Scheduler Analysis and Synthesis

The SERRA design tool is shown in Figure 23. SERRA first extracts the task control-flow from the system specification. The user-specified main task contains the overall sequence of tasks in the application. System-level task control-flow is expressed using *control-flow expressions* (CFEs) which were described briefly in Section 4. DIEGO can extract a CFE description from a task written in Verilog; for example, given the main task in Verilog, DIEGO can generate in CFE format the sequence of task invocations (calls) from the main task. Then, given the output from feeding the tasks to BC^{TM} or CINDERELLA-M, the CFEs are annotated with a $WCET$ for each hardware or software task. A single rate constraint is specified in the form of invoking the main task at a fixed rate.

SERRA synthesizes the control-unit of the scheduler by means of tool THALIA2 which takes as input a CFE description and produces a logic-level description in synthesizable Verilog[10, 29]. The timing, resource and precedence constraints specified in the CFEs input to THALIA2 are translated into a finite-state machine implementation if the constraints are satisfiable.

The constructive heuristic scheduling algorithm is implemented by CLARA, which generates the static priorities for the software- and hardware-tasks. SERRA synthesizes the control-unit of the scheduler into a hardware FSM which includes the additional precedence constraints found by CLARA.

CLARA can effectively handle multiple $NEVER$ sets, split tasks (Section 6.5), and *noninterruptible* software-tasks (Section 6.6). Furthermore, CLARA can generate precedence constraints among software-tasks in a single $NEVER$ where lower priority software-tasks can execute during idle time when higher priority software tasks are not yet ready. The analysis for this case is also implemented by CLARA, and thus it calculates $WCET$ for the out-of-order execution using the results from Section 6.4.1.

To generate the run-time kernel's C code, SERRA 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

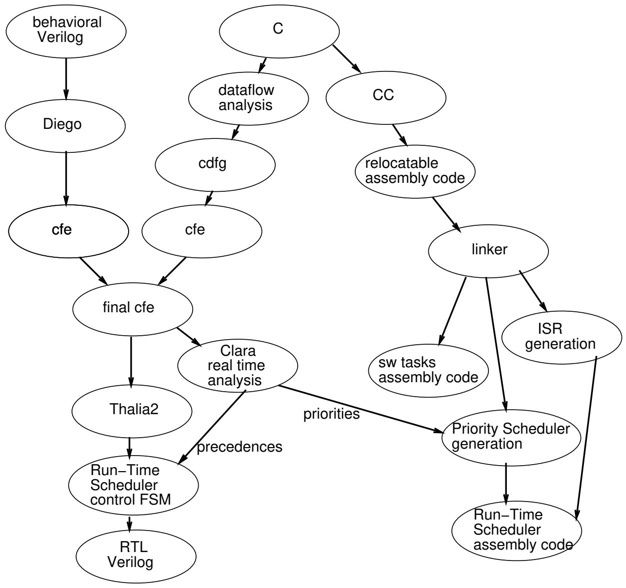


Figure 23: Block diagram of SERRA

scheduler, ISR, and context switch code using standard C compilers and linkers. Data and program memory are statically allocated.

SERRA also allows the user to override the priorities found by the heuristics of CLARA. Even further, SERRA allows the user to override precedences added to the hardware portion of the run-time scheduler, so that different software-tasks can be allowed to execute early in a different order than that found by the heuristic of Section 6.4. Thus, possible optimizations can be added by the user. SERRA can then calculate the new $WCET$ for the application with the new set of priorities and/or new set of precedences. SERRA thus provides for interactive performance evaluation of the run-time system, as well as synthesis for each particular implementation.

8 Example and Experimental Results

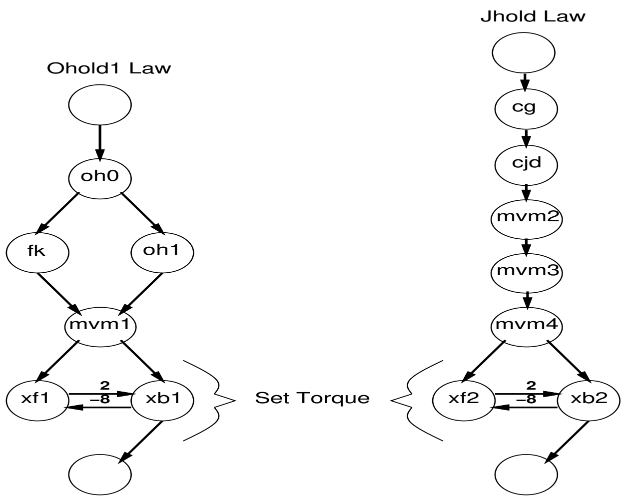


Figure 24: Directed Acyclic Graphs of Ohold1 Law, Set Torque, and Jhold Law with Relative Timing Constraints

In this section, we present an example of how a design can be successfully synthesized using the system described in the previous sections. We consider a robot controller design for manipulating two PUMA arms containing concurrent "laws" that must calculate new torques every millisecond. We show how real-time constraints can be satisfied with a run-time system that also provides for dynamic allocation of resources.

For our example, we consider the robot control algorithm of Figures 5 and 6. We implement the tasks required for executing `Jhold Law` and `Set Torque` in parallel with `Ohold1 Law` and `Set Torque`. The DAGs, including the leaf tasks that implement `Set Torque`, are shown in Figure 24. 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 our CPU in Figure 22 has an L1 cache). We assume that the full system drives `Xmit Bit` from hardware modules other than `Xmit Frame` and thus the two hardware tasks, although tightly coupled, must be kept separate.

We perform real-time analysis using the CLARA tool which has been implemented in 15,000 lines of C. We first use Constructive Heuristic Scheduling for multiple $NEVER$ sets and find the order of (`oh0`, `cjd`, `oh1`) for the software-tasks. Even with task splitting applied to `oh1`, the order does not change. Thus, we set the static priorities in the software scheduler such that $\Pi(\texttt{oh0}) > \Pi(\texttt{cjd}) > \Pi(\texttt{oh1})$. Then we run the *Execute Out-of-order* algorithm and find, just as we did in Example 24, that we should allow task `oh1` to execute on the CPU as soon as `oh0` is finished. Therefore we find the following precedence constraints: $\{\texttt{oh0} \rightarrow \texttt{cjd}\}$ and $\{\texttt{oh0} \rightarrow \texttt{oh1}\}$.

The Constructive Heuristic Scheduling algorithm found order (`mvm2`, `mvm3`, `mvm4`, `mvm1`) for $NEVER2$ and order (`xf2`, `xb2`, `xf1`, `xb1`) for $NEVER3$. Excluding redundant precedence constraints already present in the DAG, we find the following additional precedence constraints: $\{\texttt{mvm4} \rightarrow \texttt{mvm1}\}$ and $\{\texttt{xb2} \rightarrow \texttt{xf1}\}$.

As in Example 25, we calculate a $WCET$ for of 42,113 for Figure 24 with out-of-order execution. This provides for the upper bound on execution speed for the tasks in Figure 24 under worst-case conditions.

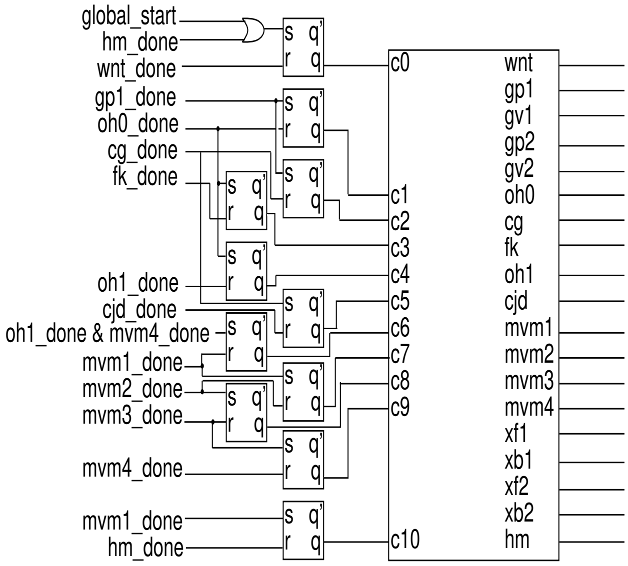


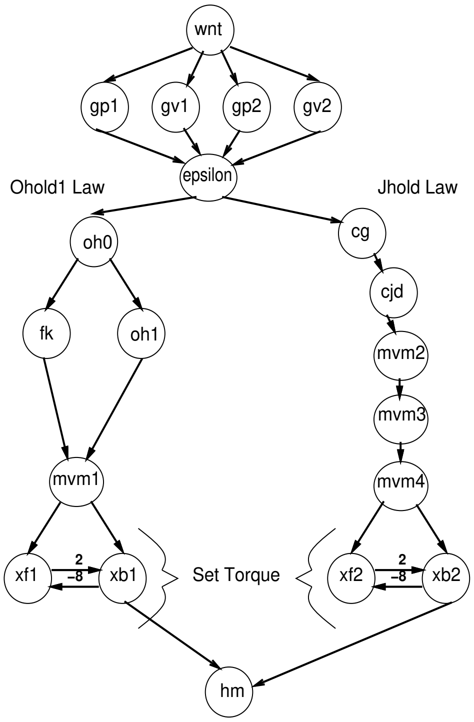
Figure 25: Final Hardware Portion of Run-Time Scheduler

Figure 25 shows the hardware portion of the run-time scheduler. Signals `wnt`, `gp1`,`gv1`,..., `hm` in Figure 25 are the *start* events for the corresponding tasks in Figures 24 and 26. The signal `global_start` kicks of execution for the very first time; after that, the *done* signal of `hm` restarts the iteration. The right-hand box is the FSM generated from the CFE for the system [10, 30]. Note that Figure 25 shows an optimization in the control logic for `mvm1`. Since the *best case execution time*, or $BCET$, of `oh1` is greater than the $WCET$ of `fk`, we can set the *start* signal of `mvm1` based only on the *done* signals of `oh1` and `mvm4` (rather than a conjunction of the *done* signals of `fk`, `oh1` and `mvm4`). Similarly, due to the length of `mvm1-4`, we find that we do not need to add the {xb2 → xf1} precedence constraint. Finally, note that the designer knows that `hm` does not need to wait for the transmission of the torque values to the robot arms; it can begin calculating right after `mvm4` finishes. These optimizations were added in SERRA manually by the user.

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, a priority scheduler which selects which software-task to execute, and routines for saving and restoring context.

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 `cg` in hardware for `Jhold Law` and `oh0` in software for `Ohold1 Law`. It continues with interleaved hardware-software execution as shown in Table 7 and pictured graphically in Figure 15. Finally, it tightly schedules accesses to `Xmit Frame` and `Xmit Bit` to set the torques for the robot.

Notice that from the point of view of the run-time scheduler, *xf1* and *xf2* are only one cycle actions; we do not wait for any *done* signal, but assume that if *xb1* completes then *xf1* has completed, and similarly that if *xb2* completes then *xf2* has completed. This was a design decision made up front based on the Verilog code for the tasks. On the other hand, notice that *xf1*, *xb1*, *xf2*, and *xb2* are all in the same *NEVER* set. This is because the same hardware-tasks, `Xmit Frame` and `Xmit Bit`, are used to transmit the torque data, and we do not want *xf2* to begin while *xb1* is still executing, nor *xf1* to begin while *xb2* is still executing. Thus we need to pay attention to the *done* events of *xb1* and *xb2*.



NEVER1 = {oh0,oh1,cjd}
NEVER2 = {mvm1,mvm2,mvm3,mvm4}
NEVER3 = {xf1,xb1,xf2,xb2}

task	task BCET (cycles)	task WCET (cycles)	icache refill WCET (cycles)
cg	11,000	11,000	
oh0	1,598	2,221	612
oh1	12,341	17,399	4050
fk	4,500	4,500	
cjd	9,989	13,213	3258
mvm1	4,400	4,400	
mvm2	4,400	4,400	
mvm3	4,400	4,400	
mvm4	4,400	4,400	
xf1	1	1	
xf2	1	1	
xb1	322	322	
xb2	322	322	
src	0	0	
sink	0	0	

Figure 26: DAG of Robot Arm Controller with Relative Timing Constraints

The complete, flattened DAG with relative timing constraints is shown in Figure 26 (reprinted from Figure 6 with additional information added). The `epsilon` task takes zero cycles and serves to synchronize the task executions by making sure every task before it has completed before continuing. The scheduling of tasks shown in Figure 26 but not in Figure 7 – `wnt`, `gp1`, `gv1`, `gp2`, `gv2`, `xf1`, `xb1`, `xf2`, `xb2`, `hm` – together take 57,200 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 42,800 cycles left for the remaining tasks – `oh0`, `oh1`, `fk`, `cg`, `cjd` and `mvm1-4`. The *WCET* of 42,113 we found fits our rate constraint (note that without out-of-order execution, we would have had a *WCET* of 46,284, which would violate the constraint). Thus, our schedule guarantees that we meet our hard real-time rate constraint.

Software-Task	Lines C	Lines Assem.	Task *BCET*	Task *WCET*	Icache refill *WCET*
oh0	90	237	1,598	2,221	612
oh1	693	3,263	12,341	17,399	4,050
cjd	286	1,177	9,989	13,213	3,258
int-ser-routine	N/A	26	11	38	N/A
context-switch	N/A	42	34	162	N/A
priority-sch-sw	107	141	26	98	N/A

Table 8: Code space, *BCET* and *WCET* for sw-tasks.

Hardware-Task	Lines Verilog	Area	*BCET*	*WCET*
cg	2897	59,587	11,000	11,000
fk	2362	42,168	4,500	4,500
mvm	629	33,645	4,400	4,400
xmit-frame	108	987	322	322
xmit-bit	66	199	322	322
run-time-sch-hw	484	413	N/A	99,701

Table 9: Results for the synthesis of hw-tasks.

Table 8 presents the results for the compilation of the software and best- and worst-case execution time estimation with CINDERELLA-M. Unfortunately, CINDERELLA-M does not perform any data-cache analysis, so all data references are assumed to miss, incurring the cost of loading in a data cache line.

In Table 9, we see the results for the synthesis of the hardware tasks of Figure 7 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 9 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, with its synthesized run-time scheduler, in Verilog using Chronologic's VCS^{TM}.

9 Conclusions

The SERRA Run-Time Scheduler Synthesis and Analysis 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.

We have addressed the important problem of real-time analysis in hardware/software co-design with a custom run-time system. The SERRA Run-Time Scheduler tool, which encapsulates the CLARA Real-Time Analysis tool, helps designers perform system-level design quickly and efficiently. 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.

Acknowledgments

This research was sponsored by ARPA under grant No. DABT63-95-C-0049 and by a Fellowship from National Semiconductor. We also acknowledge the contributions of Sera Linardi, who ported CINDERELLA to MIPS, Toshiyuki Sakamoto, who wrote the Verilog hardware-tasks and implemented interrupts in the MIPS R4K model, Firdaus Abdullah, who implemented the full Verilog simulations of the hardware-software run-time scheduler for the Robot Arm Controller, and Yau-Tsun Steven Li, who provided guidance and support for CINDERELLA-M and associated analysis.

References

[1] M. Abid, A. Changuel and A. Jerraya, "Exploration of Hardware/Software Design Space through a Codesign of Robot Arm Controller," *European Design Automation Conference,* pp. 42-47, September 1996.

[2] 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.

[3] 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.

[4] 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.

[5] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems The Polis Approach,* Kluwer Academic Publishers, Norwell, MA, 1997.

[6] 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.

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

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

[9] 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.

[10] 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, and Technical Report CSL-TR-96-694, http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs%2fCSL-TR-96-694, Stanford, CA, April, 1996

[11] 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.

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

[13] B. Dave, G. Lakshminarayana and N. Jha, "COSYN: Hardware-Software Co-synthesis of Embedded Systems", *Proceedings of the* 34^{th} *Design Automation Conference,* pp. 703-708, June 1997.

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

[15] G. De Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw Hill, Inc., New York, NY, 1994, pp. 208-211.

[16] 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.

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

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

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

[20] 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.

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

[22] M. Humphrey, G. Wallace and J. Stankovic, "Kernel-Level Threads for Dynamic, Hard Real-Time Environment," 16th. IEEE Real Time Systems Symposium, pp. 38-48, 1995.

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

[24] David C. Ku and Giovanni De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints,* Kluwer Academic Publishers, Norwell, MA, 1992.

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

[26] 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.

[27] Y. Li, S. Malik and A. Wolf, "Performance Estimation of Embedded Software with Instruction Cache Modeling", *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design,* pp. 380-387, November, 1995.

[28] 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.

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

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

[31] 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/ACM International Conference on Computer-Aided Design,* pp. 605-612, November, 1997.

[32] S. Narayan, F. Vahid and D. Gajski, "System Specification with the SpecCharts Language," *IEEE Design & Test of Computers,* pp. 6-13, December 1992.

[33] S. Prakash and A. C. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, Vol. 16, pp. 338-351, December, 1992.

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

[35] 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.

[36] 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.

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