HARDWARE/SOFTWARE CO-DESIGN OF RUN-TIME SYSTEMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Vincent John Mooney III

June, 1998

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Oyekunle A. Olukotun(Associate Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Robert W. Dutton

Approved for the University Committee on Graduate Studies:

Abstract

Trends in system-level design show a clear move towards core-based design, where processors, controllers and other proprietary cores are reused and constitute essential building blocks. Thus, areas such as embedded system design and system-on-a-chip design are changing dramatically, requiring new design methodologies and Computer-Aided Design (CAD) tools.

This thesis presents a novel system-level scheduling methodology and CAD environment, the SERRA Run-Time Scheduler Synthesis and Analysis Tool. Unlike previous approaches to run-time scheduling, we split our run-time scheduler between hardware and software, as opposed to placing the scheduler all in one or the other. Thus, given an already partitioned input system specification in an HDL and a software language, SERRA automatically generates a run-time scheduler partly in hardware and partly in software, for a target architecture of a microprocessor core together with multiple hardware cores or modules.

A heuristic scheduling algorithm solves for priorities of software tasks executing on a single microprocessor with a custom priority scheduler, interrupt service routine, and context switch code. Real-time analysis takes into account the split hardware/software implementation both of the scheduler and of the tasks. The scheduler supports standard requirements of both domains, such as relative timing constraints in hardware and semaphores in software.

A designer who uses the Serra CAD tool gains the advantage of efficient satisfaction of timing constraints for hardware/software systems within a framework that enables different hardware/software partitions to be quickly evaluated. Thus, a hardware/software partitioning tool could easily sit on top of Serra, which would generate run-time systems for different hardware/software partitions chosen for evaluation. In addition, Serra's more efficient design space exploration can improve time-to-market for a product.

Finally, we present two case studies. First, we show a full analysis, synthesis, and simulation of a hardware/software implementation of a robotics control system for a PUMA arm [AKB86, Uni84]. Second, we describe a sample prototype of the split run-time scheduler in an actual design, a force-feedback real-time Haptic robot. For this application, the hardware part of the scheduler was implemented on programmable logic communicating with software using a standard communication protocol.

Dedication

To my parents, Vincent John Mooney Jr. and Eulalia Maria Mooney, without whose love and encouragement throughout the years this thesis would not have been possible.

Acknowledgments

I have many people to thank for this dissertation. First and foremost, I would like to thank my advisor, Professor Giovanni De Micheli, for his keen insight in helping me choose an important Ph.D. topic and for his guidance throughout the Ph.D. There was more than one occasion where I arrived at a technical result, only to look back and marvel at his guidance in setting me upon the path that led to the solution, while avoiding many pitfalls which were crystal clear to me only in hindsight.

I would also like to thank Professor Oyekunle Olukotun for serving as my associate advisor and as a reader of this thesis. The interaction with Professor Olukotun and his research group – including Rachid Helaihel, Jeremy Levitt, Basem Nayfeh and Mike Chen – provided excellent opportunities for enriching and challenging the research ideas I followed. Similarly, Professor David Dill and his students – including Han Yang, Jeffrey X. Su and Clark Barrett – provided superb interaction without which my research would have been significantly compromised. Additional thanks go to Professors Olukotun and Dill for serving on my Ph.D. Orals Committee.

Special thanks go to Professor Robert Dutton for serving both as the Chair of my Ph.D. Orals Committee and as a reader of this thesis. I am very grateful to have such careful input from someone outside of my circle of immediate research colleagues.

As for the development of the SERRA Synthesis System, I would like to acknowledge the contributions of Toshiyuki Sakamoto, who wrote the hardware-tasks in Verilog HDL and implemented interrupts in the MIPS R4000 model, Sera Linardi, who ported CINDERELLA to MIPS, 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. I also would like to thank Meredith J. Goldsmith and Giuseppe A. Paleologo for extended discussions about the relation of the scheduling problem considered here to the work in operations research, as well as the help provided in formulating the problem in AMPL.

I also would like to thank Thoi Nguyen, Charles Orgish and Babak Moghadam for their network help, without which none of this would have been possible.

Many industry contacts have enriched this thesis tremendously: James Rowson, Shields Neely, Bill Mark, Mark Shand and Jim Ready, to name a few.

Finally, I would like to thank the members of the CAD group, the Robotics group, and my many friends I made while at Stanford. Claudionor Coelho, Luca Benini, Rajesh Gupta, Polly Siegel, Jerry Yang, David Filo, David Ku, James Smith, Luc Semeria, Tajana Simunic, Matija Siljak, Valeria Bertacco, Alessandro Bogliolo, Marco Platzner and Aneesh Koorapaty are current or former members of the CAD group who all helped me in some way, thanks. Diego Ruspini, Kyong-Sok Chang, and Oscar Madrigal of Professor Oussama Khatib's robotics group helped tremendously, as did Professor Khatib himself. Also, Bruce Romney of Jean-Claude Latombe's group helped in many ways, but I will only mention two: the PUMA robotics control software code and the TAship for EE271 Intro to VLSI. In addition, Scott Devine, Ben Werther, and Robert Bosch of Professor Mendel Rosenblum's Operating Systems group provided generous help at key moments. Finally, Oskar Mencer was an important ally in the Pamette/Synopsys/Xilinx struggles. As for my close friends, many of the names have already mentioned; I will say a few, but not all, of the names not mentioned: Derek Gerlach, Ajay Kapur, Rafael Betancourt, Sonja Schuemann, Lakita Garth, Florencia Cortina, Ekua Blankson, Afua Ntiwa, Manolo Clavel, Dr. Jose Meseguer, Peter Olveczky, Noel Vitug, Mike Vroman, Anthony McCarthy, Mike Pinto and Pedro Gutierrez. Dr. Rick Reis provided mentorship for which I am extremely grateful. Last but not least, my family is to be thanked: Martha Bowers Mooney, Vincent John Mooney Jr., Eulalia Maria Mooney, Alexander Xavier Mooney, Patrick Joseph Mooney, Laurie Jean Mooney, Emily Marie Mooney, Justin Patrick Mooney, and Margaret Ann Mooney.

This research was sponsored by ARPA, under grant No. DABT 63-95-C-0049, by a fellowship from National Semiconductor, and by a software donation from Synopsys.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of Computer-Aided Design (CAD) and synthesis tools in system-level design of digital systems has gained large acceptance in industry and academia. Synthesis/CAD tools automate portions of the design process, allowing designers to spend more time at higher levels of abstraction. Thus, synthesis tools support (i) more efficient exploration of the available design space, (ii) the production of correct and optimal (in some sense) circuits, and (iii) a reduction in time-to-market. These three characteristics drive the increasing use of synthesis tools in design today.

Additionally, several important trends in system-level design affect the use of synthesis. First of all, there is a significant movement towards core-based design, where pre-designed cores – such as processor and microcontroller cores – are used in system designs. Such core usage adds value through design reuse and the selling of Intellectual Property (IP). Why reinvent a component which another design team has already spent six months or more optimizing? Instead, both time and effort is saved by purchasing the component as IP.

A second trend in system-level design is the move to System-on-a-Chip (SoC) designs. For example, instead of making a board with eight separate chips, why not place all eight designs on the same chip? With ever decreasing transistor sizes, this is a real possibility for more and more systems. However, placing a system on a single Integrated Circuit (IC) requires the integration of many heterogeneous components, such as digital, analog and memory.

A third trend to help deal with the enormous complexity is the move towards standards. This is evident in several initiatives, including the Virtual Sockets Interface Alliance (VSIA)[DF98], the European CAD Standards Initiative (ECSI)[AM98], and Reusable Application-Specific Intellectual Property Developers (RAPID)[RK98]. All three seek the establishment of open standards for the easy and reliable interfacing of cores designed by separate design teams.

The direction indicated by these trends is clear. New system-level design methodologies and CAD tools are needed.

1.1 Hardware/Software Co-Design

We consider the design of mixed hardware/software systems, such as embedded systems and robots. Most of today's hardware/software systems are designed by deciding up front what functionality will be implemented in hardware and what functionality will be implemented in software, with relatively few changes as the system design progresses. The research of hardware/software co-design targets altering this design strategy.

We aim at providing 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.

Approaches to *hardware/software co-design* of embedded systems [MS96, Mic97] 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 ([HE96] defines granularity in a slightly different way). For example, the COSYMA system (Section 2.1.1 [OBE+97]), the VULCAN system (Section 2.1.2 [Gup95]) and the POLIS system (Section 2.1.3 [BCG+97]) can be classified as *homogeneous* and *fine-grained* approaches, while the COWARE system (Section 2.1.4 [MBL+96]) and the approach of Adams and Thomas [AT95] are *heterogeneous* and *coarse-grained*. The method using the SPECCHARTS language [NVG92] supports *homogeneous* specification in VHDL with both *fine-* and *coarse-grained* partitioning. We take the *heterogeneous* and *coarse-grained* approach in this thesis.

There has been much previous work in hardware-software partitioning [Gup95, MBL+96, MS96]. 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 [COB95, MBL+96].

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.

In hardware/software co-design an important problem is the management of software routines and their coordination with hardware. An indispensable component to a system of cooperating hardware and software is a run-time scheduler. One approach to scheduling is to come up with a static schedule for hardware and software operations. However, 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. Furthermore, there may be data-dependent and memory-dependent delays in execution, especially if the software runs on a processor core with caches. Thus, in many hardware/software systems, dynamic scheduling is a necessity.

One 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, this thesis proposes implementing the time-constrained portion of the scheduler in hardware, where delays are accurately known. Thus, we present a strategy for a mixed implementation of a dynamic real-time scheduler 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.

This dissertation focuses on analysis and synthesis of a custom dynamic run-time scheduler in hardware and software for embedded applications such as robotics control. In particular:

- We present a design approach for scheduling hardware/software tasks defined at a coarse level of granularity.

- We present analysis and synthesis techniques for scheduling mutually exclusive tasks to minimize *Worst-Case Execution Time (WCET).*

- We present a Verilog simulation of a robotics system using our scheduling approach as well as a small prototype of the split run-time scheduler working in an actual robotics prototype.

1.2 Requirements for Designing Hardware-Software Systems

One common requirement for system-level design targeted for mixed hardware-software implementation is the ability to carry out complex calculations. For example, in robotics control design, state space representation of the kinematics and dynamics of the arm can involve large matrices and require significant computational power [Lat91]. This complex functionality is often coupled with real-time constraints, such as the requirement to update robot arm torque inputs to its motors one thousand times a second. If missing the deadline may result in catastrophic results, such as damage to the robot or to the user, then we have a *hard* real-time constraint; if the deadline can be missed occasionally without significant negative effects, then we have a *soft* real-time constraint.

Timing constraints can be classified into two types: *rate* constraints, specifying the rate of execution of a particular set of operations or tasks, and *relative timing* constraints, specifying the minimum and maximum time separation between two operations or tasks. Both types of constraints are typical requirements in embedded systems.

In order to design a hardware-software system at a coarse level of granularity, several steps must be completed, not necessarily in this order:

- The system must be partitioned into tasks.

- Each task needs to be allocated to hardware or to software.

- The interface between tasks needs to be synthesized.
- The tasks need to be scheduled and synchronized.

The first three steps, while extremely important, have been addressed in other research and are not dealt with in this thesis. Instead, we focus on the last step, scheduling and synchronization.

Yet another requirement is the satisfaction of resource constraints. For example, there may be limited hardware, or all software-tasks might execute on the same CPU. Thus, tasks executed on the same hardware or on the same CPU cannot be scheduled at the same time, but must be mutually exclusive: one task must stop executing before the other begins to execute on the same resource. Notice that satisfying this scheduling requirement can pose a difficult optimization problem since the same task may be needed in concurrent control flows.

1.2.1 Scheduling at Different Levels

We emphasize here the differences between scheduling in high-level synthesis and scheduling in system-level synthesis. In high-level synthesis, the main emphasis is the scheduling of operations within a basic block. Optimality of a design in high-level synthesis is usually given in terms of the optimality of the execution time in basic blocks or the cost of resources in basic blocks, such as the number of multipliers, adders or multiplexors. In system-level synthesis, on the other hand, we have to consider the interactions that cross basic block boundaries as well. When the system is partitioned in basic blocks some of the interactions of the system are converted into environmental constraints, such as relative timing constraints and precedence constraints, which should guide the scheduling tool in finding a feasible and optimal implementation. Whenever these environmental constraints cross implementation paradigms – namely, hardware and software – appropriate synchronization must be added as well.

1.3 Objectives and Contributions

In this thesis, we present a system-level scheduling methodology and CAD tool. We look at a system as a collection of tasks, where a task is a hardware module or software thread. We automate the synchronization and scheduling of tasks in hardware and in software. In order to achieve this automation, we need both analysis of the system and synthesis for the run-time scheduler implemented in hardware and in software.

Specifically, we present the following contributions to the field of scheduling for hardware-software systems:

- *Design Style for Scheduling.* We will present a simple design style for representing tasks that is independent of the tasks' implementation in hardware or in software. The representation will allow for dynamic scheduling of the tasks, where by dynamic we mean that the exact time when each task starts and finishes is not statically determined but instead is decided at run-time.

- *Co-Synthesis of a Hardware-Software Run-Time Scheduler.* We will show how, given the control-flow of the tasks in the system, we can synthesize a run-time scheduler partly implemented in hardware and partly implemented in software. Such a mixed implementation can leverage the advantages of both domains.

- *Rate-Constraint Satisfaction Analysis.* We will present techniques for analysis of the satisfaction of a single hard real-time rate constraint on the system, as is typical in robotics applications. The analysis will go hand-in-hand with the scheduling of the tasks and the generation of the run-time scheduler.

- *Resource Constraints.* We will show how our scheduler synthesis procedure satisfies resource constraints, in hardware and software, while predictably satisfying timing constraints.

- *Application to Robotics.* We will present a full analysis, synthesis, and simulation of a hardware/software implementation of a robotics control system. We will also describe a small prototype of a split hardware/software scheduler to control a force-feedback Haptic robot using a Pentium^{TM} for the software and Xilinx FPGAs for the hardware.

1.4 Thesis Outline

This chapter gives an introduction and motivation for the thesis. Chapter 2 describes some of the previous work in hardware/software co-design as well as some related work from other areas. We next give an overview of our target architecture and system-level scheduling design style in Chapter 3. The chapter also briefly describes the very small kernel running on the microprocessor core.

Chapter 4 presents the real-time analysis used to analyze whether the final system will meet its timing and resource constraints. A heuristic scheduling algorithm is described in detail, together with extensions to provide support for preemptible tasks and semaphores in software.

Chapter 5 presents two design examples and how they were solved using the scheduling approach presented in this thesis. Finally, in Chapter 6, we will present some concluding remarks and some ideas for future research.

Chapter 2

Background

From the wide array of previous research in hardware/software co-design, hardware scheduling, and scheduling algorithms for real-time systems, we examine a few representative samples which most directly impact the research of this dissertation.

2.1 Previous Hardware/Software Co-Design Systems

We will first examine two systems which focus on hardware/software partitioning, after which we will examine two other systems which provide particular environments for hardware/software co-design. The next two sections exemplify two opposite approaches to hardware/software partitioning: (1) start with a software solution and migrate parts of the specification to hardware, and (2) start with a hardware solution and migrate parts of the specification to software.

2.1.1 COSYMA

The COSYMA (**COSY**nthesis for e**M**bedded micro **A**rchitectures) system aims at speeding up software execution to meet timing constraints [EHB+96, OBE+97]. The speedup is achieved by using dedicated hardware to implement some of the functionality originally calculated by software. The original specification is in C^x, a minimum extension of the C programming language to allow parallel processes. Rate constraints are specified at the process level, while input/output timing constraints can only be handled in a few specialized cases.

The original C^x specification is compiled into an **Extended Syntax Graph** of the code, annotated with local and global data flow information. Timing information is calculated using several approaches, including profiling and symbolic analysis [EY97]. With this timing analysis, COSYMA can identify which constraints are met and which are not met with the all-software solution. Next comes partitioning.

Hardware/software partitioning occurs at the basic block level, which is seen as a manageable compromise between fine-grained (at the level of individual instructions) and coarse-grained (at the level of processes or threads) partitioning. For basic blocks implemented with software, a mixed profiling/static analysis technique is used to estimate the worst case execution time (*WCET*) of the software code [EY97].

Basic blocks that are implemented in hardware are assumed to not have any pointers. High-level synthesis is performed by the **Braunschweig Synthesis System** which produces Register-Transfer-Level (RTL) output for the *Synopsys Design Compiler*^{TM}(DC^{TM}). DC^{TM} then produces a final netlist. Several techniques are used to estimate execution time of hardware, including **list scheduling** [Mic94] and **path-based scheduling** [HE95].

Communication time is estimated based on the number of variables that need to be passed between hardware and software for a given partition. Burst-mode communication is not supported.

COSYMA uses **simulated annealing** in the partitioning process. Tens of thousands of possible hardware/software partitions are considered very quickly (less than a minute) in a typical run.

COSYMA was originally targeted to single processor with one coprocessor with shared memory communication but has recently been expanded to target multiple heterogeneous processors and coprocessors running in parallel communicating over shared memory or point to point communication. Software processes mapped to the same processor are statically scheduled.

The final output of COSYMA are the hardware blocks, statically scheduled software processes, and appropriate communication primitives in hardware and in software. If a solution is generated, it is guaranteed to meet the specified rate constraints while choosing the smallest hardware cost from among the partitions considered.

2.1.2 VULCAN

The VULCAN tool aims at reducing ASIC hardware cost [Gup95]. The reduction in hardware cost is achieved by partitioning part of the design to software. The original specification is in Hardware-C [KD90], a Hardware Description Language (HDL) which can be synthesized down to netlists with the OLYMPUS Synthesis System [DKMT90].

The Hardware-C description, with rate and relative timing constraints, is mapped to a fine-grained Control-Data Flow Graph (CDFG) intermediate representation. By fine-grained we mean that nodes in the CDFG correspond to individual computations such as arithmetic operations. This is the level at which VULCAN carries out hardware/software partitioning. At locations in the CDFG where a split between hardware and software occurs, appropriate Inter-Block Communication (IBC) vertices are added. IBC vertices for communication can be blocking, nonblocking, or buffered. In addition, software has to be generated for portions of the CDFG mapped to software.

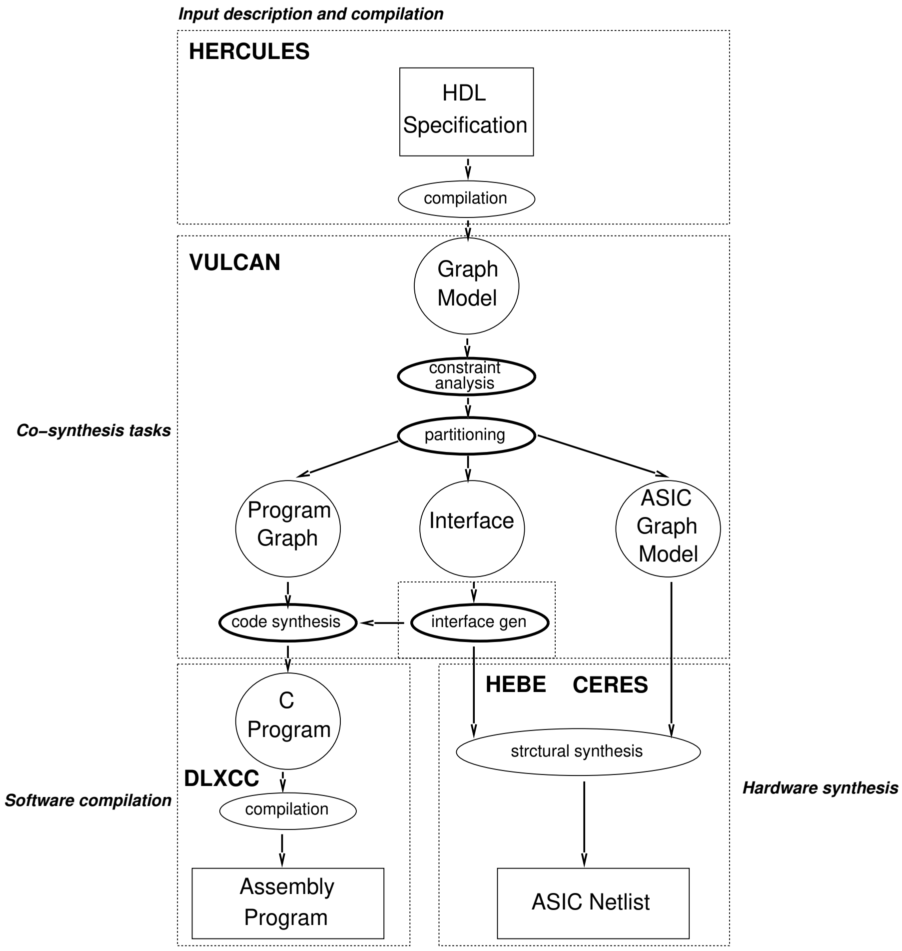


Figure 1: *Vulcan Synthesis Tool in context*

In order to map to software, all operations in the specified computation have to be serialized. Since the partial order of the CDFG specification is naturally parallel, this serialization problem is quite significant. A heuristic algorithm iterates over possible serial orders which also implement the partial order in the original specification without violating any rate constraints. The end result is a set of software program threads that can run with a custom software run-time scheduler. The run-time scheduling of software routines in VULCAN uses a non-preemptive scheme, for example as provided by a prioritized FIFO scheduler [Gup95].

Hardware/software partitioning is carried out by means of a heuristic graph partitioning algorithm which runs in polynomial time [Gup95]. The partitioning algorithm considers different partitions of the CDFG specification between hardware and software, with the goal of minimizing hardware cost while still meeting timing constraints. A graphical representation of VULCAN is shown in Figure 1.

So far we have considered, from the large amount of research, two representative systems for hardware/software partitioning. Next we will review a system for control dominated hardware/software co-design and then a system for signal processing hardware/software co-design.

2.1.3 POLIS

The POLIS system aims at providing a synthesis system targeted to design and analysis of embedded controller applications with a mixed implementation split between software and Application Specific Integrated Circuits (ASICs) [BCG+97]. The design is originally specified in a high level language such as Esterel [BG, BS91, Ber96], graphical FSMs, or Verilog/VHDL subsets.

The fundamental model of computation in POLIS is the Co-design Finite State Machine (CFSM), which supports a *globally asynchronous, locally synchronous* formal model of the design. Each transition of a CFSM takes non-zero time, is atomic, and can take on any value from a set of finite values. The assumption of non-zero transition time is made to avoid the composition problem of Mealy machines, due to undelayed feedback loops. Communication between CFSMs is by means of *events* which may be dropped (response to individual events is not guaranteed by POLIS).

In POLIS, the original design specification is compiled to a network of CFSMs. Sub-networks of CFSMs are targeted to hardware or to software; automatic synthesis supports either choice. Hardware synthesis is achieved by generating a synthesizable

HDL description and passing it on to a logic synthesis tool. Interfaces between hardware, software, and the external world are automatically synthesized in the form of cooperating circuits and software I/O drivers. For software executing on the same processor, a custom scheduler (round-robin, static cyclic, or static priority) can be compiled together with the CFSM-generated C-code, or a commercial Real-Time Operating System (RTOS) can be added by hand. Co-simulation is provided using the Ptolemy environment [BHLM94].

Thus, POLIS provides an environment where a designer can quickly evaluate choices of hardware/software partitioning, architecture selection, and scheduler selection. The output of POLIS is the C-code for the selected processor and the optimized hardware. This can be used, for example, in a board level prototype where the hardware is implemented with Field-Programmable Gate Arrays (FPGAs). The POLIS system is publicly available and has been used on several sample designs, such as a dashboard controller.

2.1.4 COWARE

The goal of the CoWare system is to provide a design environment for heterogeneous hardware/software Digital Signal Processing (DSP) systems [MBL+96, VRBM96]. CoWare was developed at IMEC Belgium, and is the basis for a commercial product CoWare N2C [Cow98]. We describe here the original CoWare, based on published research papers and presentations at international workshops and conferences [MBL+96, VRBM96, VLM96a, VLM96b, RVBM96, MBL+97]. The CoWare hardware software co-design environment allows the cospecification of hardware and software components using existing languages such as VHDL, Data Flow Language (DFL)[WDC+94], Silage and C. CoWare provides unambiguous specification of interfaces between hardware and software, and correct synthesis of these interfaces in hardware and software by generating both hardware interfaces and device drivers.

CoWare is based on a data model of communicating processes and supports the gradual refinement of a high level description into an interconnection of programmable processors and dedicated, synthesizable hardware. The model supports the re-use and encapsulation of hardware and software by a clear separation between the functional behavior and the communication behavior of a system component.

The current version of CoWare supports the use of the ARM processor and various software tools such as a simulator and compiler for ARM and commercial VHDL simulators, logic synthesis and DSP synthesis tools.

The basis for this specification method is a data model for communicating processes. The model supports a strict separation between functional and communication behavior. Designs are made reusable by describing their functional behavior while maintaining an abstract model of their communication behavior. When a design is actually (re-)used in a system, the specification method allows one to refine the abstract communication model into a detailed behavior that is more appropriate in the system context. The same specification method is used to model off-the-shelf programmable processors, and these models are used in a processor independent hardware/software co-design methodology.

Synthesis tools and compilers are able to implement all processor, accelerator, and memory components once the global system architecture has been defined. The CoWare design environment provides for integration of existing design technology by automatically generating the interfaces that link these design environments and by interfacing the generated and off-the-shelf processors in a way that is consistent with the system specification.

Designing a system with the CoWare environment involves four steps: *functional specification, architecture definition, communication selection,* and *component implementation* [VRBM96].

Functional specification. A system is specified by means of communicating processes that exchange data via channels. The behavior of a process can be entered using a host language such as C, DFL or VHDL.

Architecture definition. Optimally allocate processors, accelerators and memories, binding them to the functional specification. This interactive allocation and binding step includes the hardware/software partitioning.

Communication selection. Automatically generate the necessary software and hardware to make processors, accelerators and the different environments communicate. This step is performed via the SYMPHONY interface synthesis toolbox. Communication Blocks (CBs) provide pipelining and synchronization between accelerators. The communication between the hardware and the software for the ARM processor is more complex. The ARM interface includes address decoders, DMA channels, interrupts and I/O ports. Within the ARM, software drivers must be synthesized and linked to the processes running on the ARM.

Component implementation. All components in the system such as accelerator processors, interface hardware and software, memories, software running on a processor core, and debugging blocks are implemented using existing design environments. CoWare embeds different component compilers into the design environment, such as the ARM C-compiler and commercial VHDL/DSP synthesis environments.

The basic model of communication in CoWare is the *Remote Procedure Call* (RPC). An example can be seen in Figure 2. The RPC connections can be seen between blocks. The cascaded blocks show different abstraction levels of the same functionality. The "abstract CoWare C" is C code written for CoWare and not targeted to any particular processor. "C for CPU" is C code targeted to a particular processor, e.g. an ARM. Finally, RTL is a Register-Transfer Level description in some HDL, typically Verilog HDL or VHDL. Notice that any RPC connection can communicate with an RPC connection at any other level of abstraction – abstract CoWare C, C for CPU, or RTL.

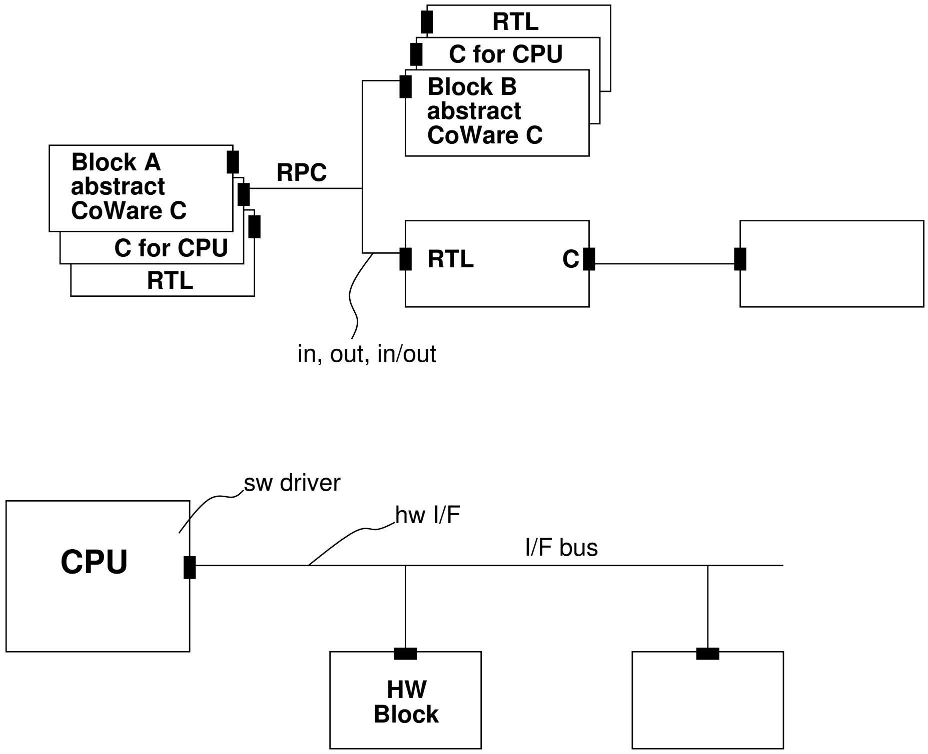


Figure 2: CoWare simulation paradigm and sample implementation

The bottom half of Figure 2 shows a hardware implementation of the RPC communication paradigm. CoWare synthesizes the software device driver as well as the logic in hardware to read data from the interface (I/F) bus.

Thus, a mixed system level specification in which part of the system is already implemented while another part is still specified at the behavioral level can be co-simulated. For this purpose, existing simulators can be integrated into the environment. Currently, Synopsys' VSS simulator for VHDL and the ARM instruction set simulators (both instruction accurate and cycle accurate) have been linked [MBL+97].

CoWare operates very much like a linker, providing an executable that can be linked to instruction set simulators as well as other modules. The CoWare methodology imposes increased demands on the generation of library elements. Often, abstract and detailed models of IP blocks do not exist. Existing IP blocks, for which

Verilog HDL code currently exists, require additional work to generate validated abstract CoWare C models.

2.1.5 CHINOOK

The Chinook system aims at providing automated interface synthesis within a hardware/software co-design framework for embedded systems [BCO96, COB95, CB94, CWB94, COB92]. A single specification language, e.g. Verilog HDL, contains both behavioral and structural descriptions of the application, including information about the processors, peripheral devices, and communication interfaces that will be used. Parts of the behavioral specification are *tagged* for preferred implementation in a particular processor or dedicated hardware, with any untagged specification assumed to be implemented in software. All interactions with the devices and interfaces are specified using a procedural abstraction layer.

Process scheduling in Chinook is achieved by assigning different *modes* of operation to the overall system. A different schedule is associated with each mode. Timing watchdogs can disable modes and cause mode transitions. Upon changing to a new mode, the system starts running the corresponding schedule. Timing constraints may be intermodal or intramodal. Each mode has a periodic set of tasks, which is unrolled and scheduled under timing constraints, using an extension of the relative scheduling formulation [KM92]. With this scheduling technique, Chinook supports the mapping of an embedded system model to one (or more) processor and peripherals while ensuring the satisfaction of timing constraints.

Chinook synthesizes device drivers, interface logic, and bus logic necessary for communication among hardware and software. For processors with general purpose I/O ports, a heuristic allocates the ports to minimize interface logic; otherwise, memory-mapped I/O is used, which includes allocating address spaces. Knowledge about the interfaces of processors and devices, which Chinook needs to carry out the synthesis, is captured in libraries.

New efforts in the CHINOOK system emphasize distributed architectures [HB97, OB97].

2.2 Hardware Scheduling

In this section we will briefly discuss some of the scheduling approaches used in high-level synthesis of hardware. In this case we have a model containing a set of operations and dependencies. The hardware implementation is assumed to be synchronous, with a given cycle-time. Operations are assumed to take a known integer number of cycles to execute. (We will later consider removing this assumption.) The result of scheduling, i.e., the set of start times of the operations, is just a set of integers. The usual goal is to minimize the overall execution *latency*, i.e. the time required to execute all operations.

2.2.1 Integer Linear Programming

The scheduling problem can be cast as an integer linear program (ILP) [Mic94], where binary-valued variables determine the assignment of a start time to each operation. Linear constraints require each operation to start once and to satisfy the precedence and resource constraints. Latency can also be expressed as a linear combination of the decision variables. The scheduling problem has a dual formulation, where latency is bounded from above and the objective function relates to minimizing the resource usage, which can also be expressed as a linear function. Timing and other constraints can be easily incorporated in the ILP model.

The appeal of using the ILP model is due both to the uniform formulation, even in presence of different constraints, and to the possibility of using standard solution packages. Its limitation is due to the prohibitive computational cost for medium-large cases. This relegates the ILP formulation to specific cases, where an exact solution is required and where the problem size makes the ILP solution viable.

2.2.2 List Scheduling

Most practical implementations of hardware schedulers rely on *list scheduling*, which is a heuristic approach that yields good (but not necessarily optimal) schedules in linear (or overlinear) time. A list scheduler considers the timeslots one at a time, and schedules to each slot those operations whose predecessors have been scheduled, if enough resources are available. Otherwise the operation execution is deferred. Ties are broken using a priority list, hence the name.

2.2.3 Relative Scheduling

The *synchronization* of two or more operations or processes, often with exact cycle minimum and maximum separation timing constraints, is an important issue in hardware scheduling. Synchronization is needed when some delay is unknown in the model – the assumption that all operations take a known integer number of cycles to execute is removed. *Relative scheduling* is an extended scheduling method to cope with operations with unbounded delays [KM92] called *anchors*. The presence of an anchor means that a static schedule cannot be determined. Nevertheless, in relative scheduling the operations are scheduled with respect to their anchor ancestors. A FSM can be derived that executes operations in an appropriate sequence, on the basis of the relative schedules and the anchor completion signals. Relative scheduling support the analysis of timing constraints; when these constraints are consistent with the model, any resulting schedule generated is guaranteed to satisfy the constraints for any anchor delay.

2.2.4 Conditional Process Graphs

A recently published paper considers the case where a Directed Acyclic Graph (DAG) specifies a set of processes with precedence constraints [EKP+98]. Each edge in the DAG may have a conditional associated with it.

The goal is to generate a static schedule which will minimize the execution time of the DAG for any allowable value of the conditionals. Since this may require activations of different tasks in different orders, they keep track of the possible paths using a *schedule table*. Alternative paths through the DAG are captured with BDDs. There may be a *conflict*, where, for example, the optimal schedule of one path requires that process P_3 be scheduled at time t_k, while the optimal schedule of another path requires that P_3 be scheduled at time t_l, $t_k \neq t_l$. Conflicts are handled by adjusting one of the path schedules.

This technique is applicable to hardware/software systems. The end result is a distributed run-time scheduler composed of non-preemptive schedulers. Conditionals are broadcast so that individual schedulers can dynamically choose the appropriate schedule for the processes under their control. For the case where each process can be allocated either to hardware or to a programmable processor, then this scheduling technique applies to hardware/software co-design.

2.3 Software Scheduling

We will next examine some representative examples of previous approaches to scheduling for real-time software systems. The goal is software scheduling to meet real-time constraints. In the following, the assumption is that a large scale software system, with hundreds or thousands of individual tasks with many different periods and deadlines, is being designed.

2.3.1 Round-Robin Scheduling

The round-robin scheduling algorithm takes a small slice of time and allocates each process on a circular queue the time slice. If the process takes less than the time slice to execute, then the scheduler immediately goes to the next process in the circular queue. Otherwise it preempts the currently executing process at the end of the time slice and runs the next process in the queue. As the size of the time slice approaches infinity, the round-robin policy becomes the same as the First-Come-First-Served (FCFS) policy. While this algorithm is very predictable and by design avoids starvation and deadlock, unfortunately it can result in large average waiting time and many extra context switches.

2.3.2 Shortest Job First

The shortest-job-first scheduling algorithm requires that each process have associated with it the length of uninterrupted CPU execution it needs next. This length can either be the entire length of the process or the length of the next CPU burst where it will heavily use the CPU (as opposed to waiting on I/O or for synchronization with other processes). Then, shortest-job-first assigns the CPU to whichever available process has the smallest length of uninterrupted CPU execution associated with it. When that process finishes, the CPU is assigned again to the process with the shortest length. While the shortest-job-first algorithm is optimal in terms of minimizing the average waiting time, it may result in missing timing constraints where another schedule would have met the timing constraints.

2.3.3 Rate-Monotonic Analysis

Rate Monotonic Analysis (RMA) [LL73] and Generalized Rate Monotonic Analysis (GRMA) [SRS94] both assume that tasks are independent and that each task has its own period and deadline which are the same and never change. Furthermore, each task is assumed to have a constant run-time which does not change over time. In RMA, the *rate-monotonic priority assignment* assigns higher priorities to tasks with higher priorities. Such a priority assignment has been proven optimal in the sense that no other fixed priority assignment can schedule a set of tasks which cannot be scheduled (without missing deadlines) by the rate-monotonic priority assignment [LL73]. Liu and Layland were able to prove the following theorem:

Theorem 2.1 *A set of n independent periodic tasks scheduled by the rate-monotonic algorithm will always meet their deadlines for all task start times, if*

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{\frac{1}{n}} - 1)$$

where C_i is the execution time and T_i is the period of task τ_i.

Theorem 2.1 ignores all overheads (assumed to be zero). The bound on the utilization $n(2^{\frac{1}{n}} - 1)$ rapidly converges to $\ln 2 = 0.69$ as n becomes large.

GRMA adapts the RMA framework to deal with problems typically faced in real-time software systems. For example, a typical problem is *priority inversion*. This occurs when a lower priority process holds a critical resource, thereby preventing a higher priority process from executing when it interrupts and tries to access the critical resource: the priorities of the two process have been inverted because the lower priority process has, in effect, made itself higher in priority. Thus, GRMA supports the *priority ceiling protocol*, which avoids mutual deadlock arising from the priority inversion problem [SRL90, Raj91]. The deadlock is avoided by having any lower priority process holding a critical resource inherit the priority of any higher priority processes which try to access the resource, until the lower priority process releases the resource (at which point the lower priority process resumes its original priority).

RMA has also been extended to account for release jitter and some cases of resource contention [ABD+95, ABR+93].

However RMA and GRMA both fail when the tasks have *precedence* constraints. We assume the presence of precedence constraints in the system in this dissertation.

2.3.4 Least Common Multiple

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 [Ram95, Ram90, PS89]. Unfortunately, this approach is usually impractical for hardware/software co-design for several reasons:

- First of all, 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 [YW96, YW95].

- Secondly, the task periods can be large and co-prime, resulting in a LCM too large to be practical.

- Thirdly, it discourages static allocation and scheduling because it treats different instances of the same task as different nodes in the LCM.

One approach to deal with the third problem mentioned above is to use the concept of an *association array* which keeps track of the priority level, allocation to hardware or CPU, deadline, and best/worst-case execution time for each copy of each task [DLJ97, DJ98].

2.3.5 Real-Time Kernel in Hardware

A common approach is to encapsulate software scheduling algorithms into a fast but general purpose operating system, called a *Real-Time Operating System (RTOS)*. The basic idea is to provide the functionality needed by real-time software systems without the large overhead associated with traditional operating systems. A good overview of RTOS research in scheduling algorithms is contained in [SSNB95].

One interesting RTOS research direction implements a small real-time kernel in hardware to run in parallel with multiple processors [AFLS96, LSF95, Lin92, LS91]. The real-time kernel contains a scheduler with a priority scheduling algorithm, a dispatcher which controls the task switch mechanism, a wait queue for inactive tasks, a wait queue for tasks waiting for a time event, and a ready queue. A prototype of the system contains a VME bus connecting the real-time kernel in hardware, a bus arbiter in hardware, a large RAM, and three processors. To schedule a software task on a particular processor, the kernel triggers an interrupt on the processor, which results in placing the task id of the new task in a register on the processor. The new task id is read from that register, performing a task switch. The real-time kernel can handle a maximum of 64 tasks at 8 priority levels [AFLS96].

2.4 Control-Flow Expressions

Control-flow expressions (CFEs) [CM96, Coe96, CM97] support system-level specifications in an algebraic formalism that considers most of the language constructs used to model systems reacting to their environment, i.e. sequential, alternative, concurrent, iterative, and exception handling behaviors. Such constructs are found in languages such as C, Verilog HDL, VHDL, Esterel and StateCharts. CFEs can specify control flow that satisfies relative timing constraints (minimum and maximum

Composition	HL Representation	CF Expression
Sequential	**begin** P; Q **end**	$p \cdot q$
Parallel	**fork** P; Q **join**	$p \| q$
Alternative	**if** (C) P ; **else** Q ;	$c : p + \overline{c} : q$
Loop	**while** (C) P ; **wait** $(!C)$ P ;	$(c : p)^*$ $(c : 0)^* \cdot p$
Infinite	**always** P ;	p^ω

Table 1: *Link between Verilog HDL Constructs and Control-Flow Expressions*

separation)[KM92] in hardware while also controlling dynamically the flow of execution.

2.4.1 Formalism

Input events of a control-flow are specified by *conditionals*, which enable different blocks of the specification to execute based on the input. Output control signals are specified by *actions* which control execution according to the control-flow; for example, an action becoming asserted may indicate that a multiplier should begin execution. Table 1 shows the correspondence between CFEs and standard Verilog HDL control-flow constructs.

Example 1 Suppose we have an alternative choice where based on conditional c, we execute either an adder, represented by CFE action a, or a multiplier, represented by CFE action m. The CFE description of this conditional choice, assuming that the adder and the multiplier are single-cycle actions, is as follows: $c : a + \overline{c} : m$ □

Any CFE expression can be compounded sequentially or in parallel with any other CFE expression. Furthermore, with CFEs one can represent the control flow of most digital systems.

Synchronization constraints are specified through the use of $NEVER$ and $ALWAYS$ sets. $NEVER$ sets model mutual exclusion; for example, $NEVER = \{a, b, c\}$ indicates that actions a, b, and c can never be active at that same time. In a similar vein, tasks that must begin execution concurrently 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. Thus, with conditionals, $NEVER$ sets, and $ALWAYS$ sets, CFEs can specify and consider constraints crossing concurrent blocks of the design, which are generally ignored in other synthesis tools.

Using conditional execution within a loop models an action with unbounded delay, e.g. as shown by the CFE construct $(c : p)^*$. Using a composition of these unbounded conditional executions can model a sequence of hardware and software tasks with unknown delay. Thus, with CFEs one can dynamically control the execution task, beginning a subsequent task after, and exactly after, all of its preceding tasks have finished execution.

Example 2 Suppose we want to execute the following tasks infinitely often: a followed by b, in parallel with d followed by e. Tasks a and d have nondeterministic delay. Tasks b and e take one cycle each. There are no constraints, i.e. no relative timing constraints, $NEVER$ sets, or $ALWAYS$ sets. Associating a CFE control signal and a CFE action with each task, we end up with the following expression (recall that in CFE semantics, * indicates zero or more cycles, $||$ indicates parallel execution, $\cdot$ indicates serial execution, and $^\omega$ indicates an infinite loop):
$(((c1 : a)^* \cdot b)||((c2 : d)^* \cdot e))^\omega$ $\Box$

2.4.2 Synthesis with CFEs

The synthesis procedure begins by converting the *control-flow expressions* into an automaton where design constraints such as timing, resource and synchronization are incorporated. The synthesis is conservative in the sense that a solution is produced only if the specified constraints, such as relative timing constraints, are satisfied. In order to generate the control-units for the design, two scheduling procedures are used. The first procedure, called static scheduling, attempts to find fixed schedules for operations satisfying system-level constraints. The second procedure, called dynamic scheduling, attempts to synchronize concurrent parts of a circuit description by dynamically selecting schedules according to conditionals from the rest of the system. The solution of both scheduling problems are cast as Integer Linear Programming instances and solved using Binary Decision Diagrams.

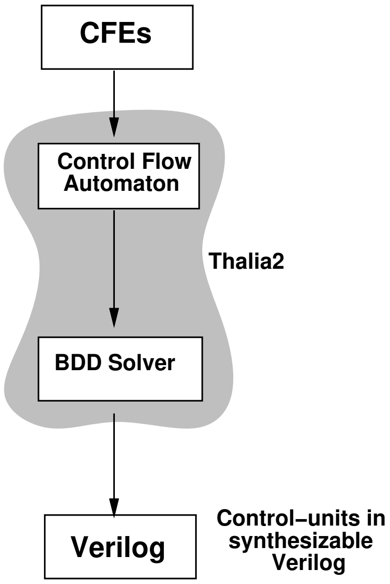


Figure 3: THALIA2 Synthesis from CFEs

2.4.3 Thalia

The algorithms to synthesize a controller from a CFE specification have been implemented in a tool called Thalia [CM96, Coe96] which outputs a logic description. This logic description can be synthesized by commercial (e.g. Synopsys) or research (e.g. Olympus[DKMT90] or SIS[SSM+92]) synthesis systems. We have modified Thalia slightly in order to output synthesizable Verilog HDL; we call the new tool Thalia2, as shown in Figure 3.

2.5 Bounds on Execution Time

Real-time systems design requires bounds on execution time of the various components. The lower bound is often referred to as the *Best-Case Execution Time (BCET)*, while the upper bound is referred to as the *Worst-Case Execution Time (WCET)*. Recent previous approaches to such real time analysis have focused on software, since the performance analysis of ASICs is considered a well studied problem already. One such approach is that of the cinderella tool [MWWL96, LM95], which this section discusses since we will use cinderella in the timing analysis presented in this thesis.

cinderella addresses the problem of determining *BCET* and *WCET* bounds for a given program executed on a given processor, assuming uninterrupted execution. Two important issues in solving this problem are (i) program path analysis, which determines which sequence of instructions will be executed in the worst case, and (ii) microarchitecture analysis, which requires modelling the hardware system to determines the *BCET* and *WCET* execution bounds for a given sequence of assembly instructions. cinderella divides the assembly code into its basic blocks and extracts the control-flow among the blocks. Thus, explicit path enumeration is not required by cinderella. Software is assumed to not have dynamic memory allocation nor recursive function calls. However, loops are allowed as long as the user can provide upper bounds for each loop specified. Then, using an ILP formulation, CINDERELLA finds a *BCET* and a *WCET* for the program in clock cycles of the microprocessor clock. Target microarchitectures include the Intel i960KB processor and the Motorola 68000 processor [MWWL96, LM95]. We have ported CINDERELLA to the MIPS R4000 processor.

2.6 Summary

We have shown some representative samples of previous work in hardware/software co-design and algorithms for real-time scheduling of software. In hardware/software co-design, we have seen several systems for hardware/software partitioning and evaluation of different hardware/software tradeoffs. Only a few of the systems considered have as a primary goal the support of heterogeneous input in both a software language and an HDL, which will be the approach we take in this thesis. Finally, some of the previous work in scheduling for hardware and for finding bounds on the execution time of software was discussed.

None of the previous hardware/software co-design systems focus on run-time scheduling, and all place the run-time scheduler on the CPU (hardware is assumed to have a static schedule implied by the input description of the application). This thesis extends previous work to support dynamic interleaving of hardware-software execution, where by dynamic we mean that exact start times are not statically determined but instead are decided at run-time based on actual execution times. To achieve this, we present the first implementation of a run-time scheduler split partly in hardware and partly in software. Furthermore, real-time analysis will be provided to determine a *WCET* for the system.

In the next chapter, we will explain the target architecture for our system, the small kernel to control software running on a microprocessor, run-time scheduler implementation details, and the approach to modelling of the system.

Chapter 3

Target Architecture, Kernel, and System Model

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. We will present the SERRA tool, which 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 model either hardware or software and to be written either in Verilog HDL 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.

3.1 CAD Requirements

Figure 4: PUMA Arms (Courtesy of the Computer Science Robotics Lab at Stanford)

Example 3 As a motivational example, consider the set of control algorithms of Figure 5. These algorithms calculate torques for the PUMA[AKB86, Uni84] robot arms shown in Figure 4.

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 [ACJ96].

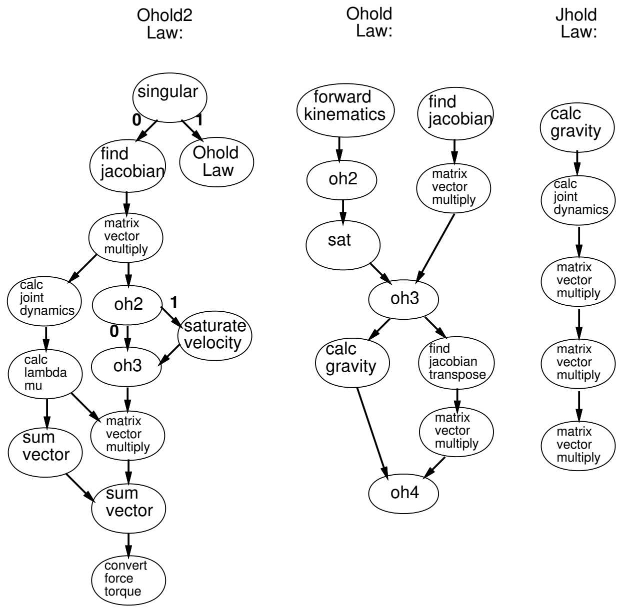


Figure 5: Robotics Example: Concurrent Control Algorithms

Figure 5 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 3 are as follows. First, we need to satisfy hard real-time constraints imposed by some of the hardware components in the system as well as by external hardware. Second, we need to optimize the run-time system over calls to multiple tasks in hardware and software. This involves allocation of tasks to hardware and software as well as interface generation for communication. 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 thesis.

We design our run-time scheduler CAD tool, which we call SERRA, to work with

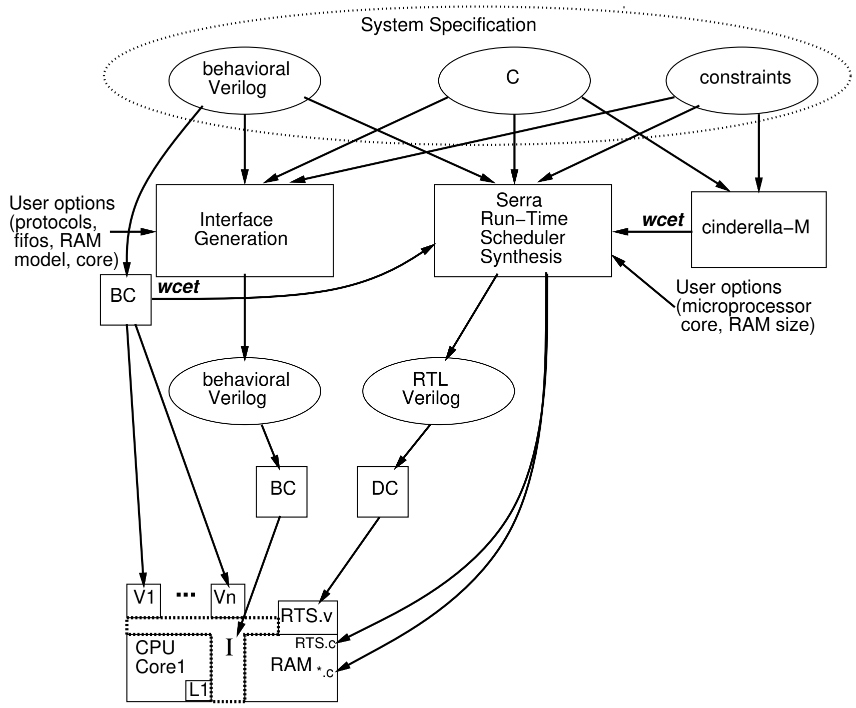


Figure 6: Tool Flow and Target Architecture

existing hardware synthesis and software compilation tools. Figure 6 shows the tool flow in which Serra is embedded. Hardware-tasks are specified in Verilog HDL that can be synthesized by the Synopsys Behavioral Compiler^{TM}[Kna96] (labeled `BC` in Figure 6; `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 HDL 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 the tool discussed in Section 2.5, CINDERELLA[MWWL96, LM95]. The new backend is for MIPS assembly run on a MIPS R4000 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})[Kna96], 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.

The run-time scheduler synthesis of SERRA 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 the serial order for hardware-tasks executed using the same hardware resource. These software-task priorities and hardware-task serial orderings are chosen to minimize *WCET* 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 HDL to speed up a critical path in the algorithm.

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

3.2 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.

3.2.1 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 other and different tasks, such as `matrix vector multiply` in our robot example, as can be seen in Figure 5. 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.

3.2.2 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.

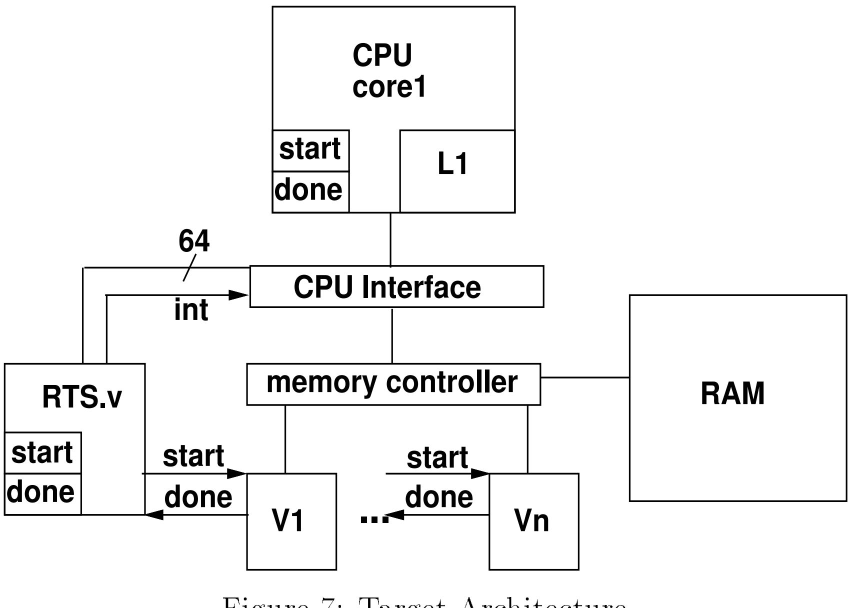


Figure 7: Target Architecture

Figure 7 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.

3.2.3 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 7 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.

3.2.4 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 shown in Table 2.

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

Table 2: 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 (fixed) 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 [MSM97]. 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 thesis we only consider a priority driven scheme.

3.2.5 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 Chapter 4. We now turn to modeling issues.

3.3 System Modeling

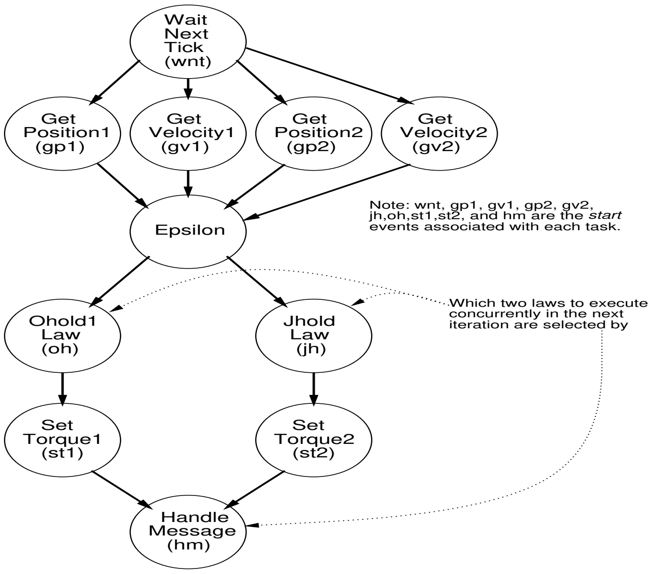


Figure 8: Robotics Example: Main Task

The input specification is a collection of tasks written in Verilog HDL 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 8). 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.

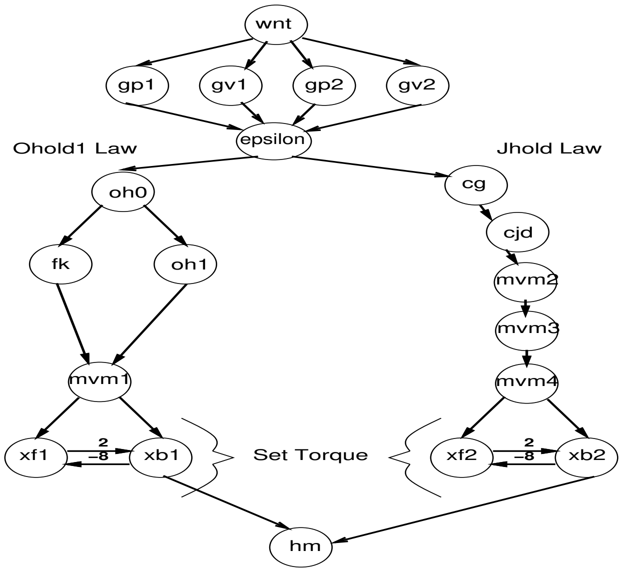


Figure 9: Flattened CDFG of Robot Arm Controller

Example 4 Figure 8 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 HDL. The other tasks are specified in C and Verilog HDL.

Note that the CDFG of Figure 8 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 9. The flattened CDFG executes an appropriate subset of the control algorithms of Figure 5 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. □

SERRA uses the Synopsys Behavioral Compiler^{TM} for the synthesis of hardware tasks. SERRA leverages previous research on system modeling using *control-flow expressions* (CFEs) [CM96, Coe96], as covered in Section 2.4. In SERRA, CFEs represent an intermediate model of the run-time system that captures the global control-flow information in the system. This contrasts with earlier uses of CFEs to model systems at the operation level [MCSM96]. Using CFEs to coordinate tasks hides the coordination of low-level operations from the CFE model and results in greatly reduced control logic. We assume that the total number of tasks in the system is around 50 to 100.

We support the specification of tasks that cannot execute concurrently through the use of CFE $NEVER$ sets. 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 thesis, 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 thesis.

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

3.4 Summary

In this section we outlined our approach and design style for hardware/software co-design of a run-time scheduler split between hardware and software. We presented a target architecture and small software kernel to manage software-tasks. Finally, we reviewed our system-level modeling of task flow for any specified application.

In the next chapter, we will focus on the analysis and synthesis of a custom run-time scheduler, which requires the satisfaction of a single rate constraint and multiple resource constraints.

Chapter 4

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 chapter, 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 4.2.3.

The outline of this chapter is as follows. In Section 4.1, we first present the assumptions we make in defining our problem and prove the resulting (decision) problem to be NP-complete. In Section 4.2 we present the Constructive Heuristic Scheduling approach to solving our problem, which is the ordering of tasks which use the same resource and thus must execute in a mutually exclusive fashion. In Section 4.2.3 we extend the heuristic of the previous section to deal with the case of multiple sets of mutually exclusive tasks. Section 4.3 presents analysis and a greedy heuristic to deal with the case of preemptible tasks (which leads to increased context switches). Finally, Sections 4.4 and 4.5 improve the heuristic and extend it to support critical regions, thus providing the same functionality as semaphores.

4.1 Assumptions and Complexity

To predictably satisfy a 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}[Kna96] as mentioned in Section 3.1. We need some assumptions to compute the *WCET* for the set of tasks.

Assumption 4.1 *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 5 Figure 9 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 branching statement is not modelled. 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.

Assumption 4.2 *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 4.3 *Each task, once started, runs to completion.*

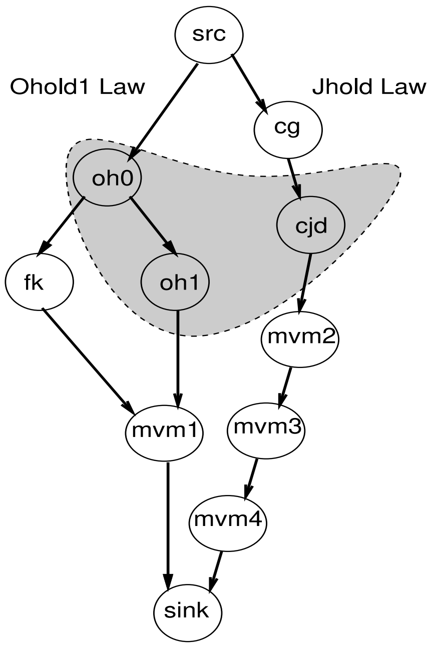
Together with the previous assumption and the fact that the `priority_scheduler` code only uses registers reserved for the operating system, we find that the only overhead for software-tasks are the ISR and `priority_scheduler` calls. We will relax the assumption of running to completion later when calculating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

Assumption 4.4 *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 [COB95, MBL$^+$96, VRBM96].

Assumption 4.5 *Interrupts that switch context come only from the hardware run-time scheduler as described in Section 3.2.3.*

Example 6 As an example, consider Figure 10. This represents a subset of the tasks from the robot controller shown in Figure 9. The *WCET* times for the individual tasks have already been calculated by CINDERELLA-M and BC^{TM}. Three tasks are specified in Verilog HDL: `mvm`, `fk`, and `cg`, corresponding to `matrix vector multiply`, `forward kinematics`, and `calc gravity`, respectively, in Figure 5. (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 5 and both `oh0` and `oh1` are coarser-grained groupings of tasks called



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 10: DAG, $BCET$ and $WCET$: The leftmost column shows the task names, the middle column shows the Best-Case Execution Time, and the rightmost column shows the Worst-Case Execution Time.

by `Ohold Law` in Figure 5. 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 10. 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. □

Example 6 shows a difficult problem in that a $NEVER$ set of software-tasks may cross parallel paths. This problem, which we refer to as the Never Set DAG Scheduling (NSDS) problem, cannot be solved with a single execution of a longest path algorithm 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* – the NSDS problem – will be shown to be NP-Hard in the next section.

4.1.1 The Complexity of NSDS

In this section we discuss the complexity of NSDS. As is customary, we cast NSDS as a decision problem. Note that in the following we distinguish between *GraphWCET*, the *Worst-Case Execution Time* for a set of tasks in a DAG, and *WCET*, the *Worst-Case Execution Time* for an individual task. We give a formal definition of NSDS-decision as follows:

Definition 4.1 [NSDS-decision] *INSTANCE: Directed Acyclic Graph* $G(V, A)$*, allowable GraphWCET of* K *for the graph, NEVER set* $N \subseteq V$*, set* T *of tasks with* $V = T$ *and, for each task* $t \in T$*, a length* $l(t) \in Z^+$*.*
QUESTION: Is there a one-processor schedule for N *that satisfies the allowable GraphWCET of* K *for the DAG, i.e. a one-to-one function* $\sigma : T \rightarrow Z_0^+$*, with, if* $t_i \in N, t_j \in N - \{t_i\}$*, either* $\sigma(t_i) > \sigma(t_j)$ *implying* $\sigma(t_i) \geq \sigma(t_j) + l(t_j)$ *or* $\sigma(t_j) > \sigma(t_i)$ *implying* $\sigma(t_j) \geq \sigma(t_i) + l(t_i)$*, such that, for all* $t \in T$*,* $\sigma(t) \geq \sigma(t_p) + l(t_p)$*, where* t_p *is a predecessor of* t *in* $G(V, A)$*, and* $\sigma(sink) + l(sink) < K$*, where sink is the sink task in* $G(V, A)$*.*

Notice that for the case of two tasks in the NEVER set, $t_i \in N, t_j \in N - \{t_i\}$, forcing either $\sigma(t_i) > \sigma(t_j)$ implying $\sigma(t_i) \geq \sigma(t_j) + l(t_j)$ to be true or $\sigma(t_j) > \sigma(t_i)$ implying $\sigma(t_j) \geq \sigma(t_i) + l(t_i)$ to be true ensures a serial order for software-tasks executed on the same processor. Note that σ records the start times for the tasks in $G(V, A)$. Also note that for each task $t \in T$, $l(t)$ is equivalent to a $WCET$ for t. For the actual NSDS problem, of course, we are not given a maximum allowable *GraphWCET* of K, but instead we try to find the minimum such K possible.

Now, to analyze the complexity of NSDS-decision, we use the Sequencing with Release Times and Deadlines (SRTD) problem[GJ79], which is defined as follows:

Definition 4.2 [SRTD] *INSTANCE: Set T of tasks and, for each task $t \in T$, a length $l(t) \in Z^+$, a release time $r(t) \in Z_0^+$, and a deadline $d(t) \in Z^+$.*
QUESTION: Is there a one-processor schedule for T that satisfies the release time constraints and meets all the deadlines, i.e. a one-to-one function $\sigma : T \rightarrow Z_0^+$, with, if $t' \in T - \{t\}$, $\sigma(t) > \sigma(t')$ implying $\sigma(t) \geq \sigma(t') + l(t')$, such that, for all $t \in T$, $\sigma(t) \geq r(t)$ and $\sigma(t) + l(t) \leq d(t)$?

The Sequencing with Release Times and Deadlines (SRTD) problem deals with the situation where the task t is executed from time $\sigma(t)$ to time $\sigma(t) + l(t)$, cannot start executing until time $r(t)$, must complete by time $d(t)$, and cannot overlap the execution of any other task t'.

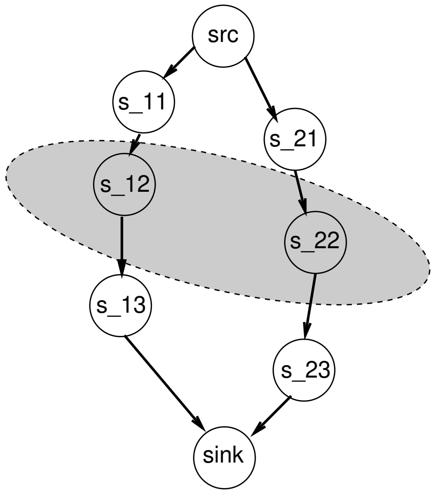
Theorem 4.1 *NSDS-decision is NP-Complete.*

Proof: First, given an order for the software-tasks (which are all in the NEVER set N), one can check in polynomial time if the $GraphWCET$ of the DAG is less than K.
Next, SRTD can be reduced to NSDS-decision as follows.
Let an instance I_{SRTD} of SRTD be given with $T = \{t_1, t_2, \ldots, t_n\}$. Define K to be equal to $(\sum_{t_i \in T} l(t_i)) + \max_{t_i \in T} r(t_i)$. (Note there is no need to add any $d(t)$ to K, since $d(t)$ is just a deadline.) Now we define an instance $I_{NSDS-decision}$ of NSDS-decision by assigning nodes in NSDS-decision for each task $t_i \in T$ as follows:

- let s_{i1} be a hardware-task with WCET equal to $r(t_i)$, where the predecessor of s_{i1} is the source and the successor of s_{i1} is s_{i2}
- let s_{i2} be a software-task with WCET equal to $l(t_i)$, where the predecessor of s_{i2} is s_{i1} and the successor of s_{i2} is s_{i3}

- let s_{i3} be a hardware-task with WCET equal to $K - d(t_i)$, where the predecessor of s_{i3} is s_{i2} and the successor of s_{i3} is the sink (note that if $d(t_i) > K$, then we remove s_{i3} from the DAG)

Basically, s_{i1} enforces the release time constraint and s_{i3} enforces the deadline. Now, place all software-tasks in a single NEVER set. Clearly, I_{SRTD} has a solution if and only if $I_{NSDS-decision}$ has a solution.
QED. ♯



NEVER = {s_12,s_22}

task	wcet (cycles)
s_11	0
s_12	3
s_13	4
s_21	2
s_22	4
s_23	3

Figure 11: Example transformation of an SRTD problem to an NSDS problem.

Example 7 Consider an instance of SRTD with two tasks: task t_1 with length $l(t_1) = 3$, release time $r(t_1) = 0$, and deadline $d(t_1) = 5$; and task t_2 with length $l(t_2) = 4$, release time $r(t_4) = 2$, and deadline $d(t_1) = 6$. Then, we calculate $K = 3+4+2 = 9$. We add the following tasks to the NSDS instance to account for t_1 in SRTD: s_{11} with a $WCET$ of 0, s_{12} with a $WCET$ of 3, and s_{13} with a $WCET$ of $K - d(t_1) = 9 - 5 = 4$. Next, we add the following tasks to the NSDS instance to account for t_2 in SRTD: s_{21} with a $WCET$ of 2, s_{22} with a $WCET$ of 4, and s_{23} with a $WCET$ of $K - d(t_2) = 9 - 6 = 3$. The two tasks s_{12} and s_{22} are placed in a single $NEVER$ set. Figure 11 shows the resulting NSDS instance. □

Trivially, since NSDS-decision is NP-Complete, NSDS is NP-Hard (at least as hard to solve as an NP-Complete problem) [GJ79]. In the context of our system design, solving the NSDS problem allows us to proceed with our real-time analysis. For example, once we have a *WCET* for the CDFG of Figure 8, then we can say if the robot controller finishes execution within one millisecond.

4.2 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[HL95]. 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 4.2.3, we will extend constructive heuristic scheduling to include multiple resource constraints in the form of $NEVER$ sets of hardware-tasks.

4.2.1 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 4.3 *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 4.4 *Let t denote a task, and let t_i denote a task executed in stage i.*

Definition 4.5 *Let the multivalued decision variables x_{ik}, $i \in (1, 2, \ldots, n-1)$ and $k \in Z^+$, 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 4.6 *Let X_i, $i \in (1, 2, \ldots, n-1)$, denote the multiset of decision variables $\{x_{ik}\}$.*

Example 8 Consider Figure 10. Since $|NEVER| = 3$, there are 5 stages. In stage 3 we could find that $X_3 = \{x_{31}, x_{32}, x_{33}\} = \{(\texttt{oh0},\texttt{sink}),(\texttt{oh1},\texttt{sink}),(\texttt{cjd},\texttt{sink})\}$. Each x_{3k} is an ordered set, and X_3 is a multiset. □

Definition 4.7 *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 9 In Figure 10 the tasks under consideration are `src`, `oh0`, `oh1`, `cjd`, and `sink`. Since the `sink` is always executed last, $X_{n-1} = X_4 = \{(\texttt{sink})\}$. The possible tasks executed before the `sink`, and thus in stage 4, are $t_4 = \texttt{oh1}$ and $t_4 = \texttt{cjd}$. Thus the possible states in stage 4 are $s_4 = (\texttt{oh1},\texttt{sink})$ and $s_4 = (\texttt{cjd},\texttt{sink})$. □

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

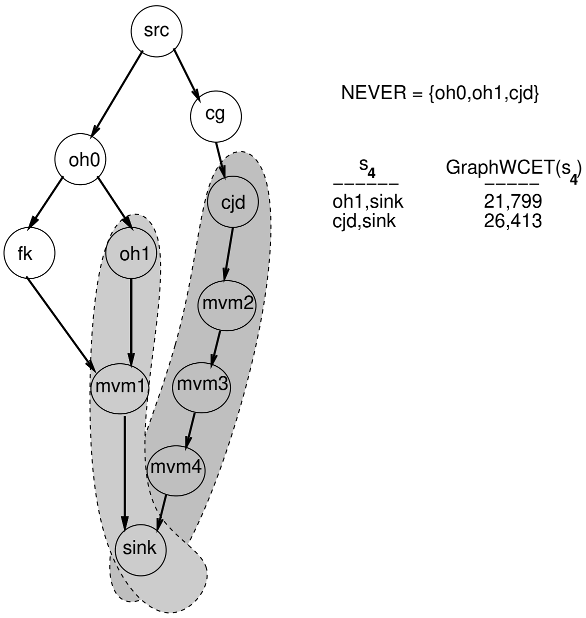


Figure 12: *GraphWCET* Example

Definition 4.8 *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 10 Consider Figure 12. In this example we are in stage $i = 4$. The leftmost shaded area covers G_{s_4} defined by $s_4 = (\texttt{oh1,sink})$. In this case $V_{s_4} = \{\texttt{oh1,mvm1,sink}\}$. □

Definition 4.9 *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 11 Consider Figure 12 again. The two valid states in stage 4 are $s_4 = (\texttt{oh1,sink})$ and $s_4 = (\texttt{cjd,sink})$. State $s_4 = (\texttt{oh0,sink})$, however, is not a valid state because G_{s_4} contains oh1, which is in the $NEVER$ set but not in s_4. □

Definition 4.10 *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 12 Continuing with Figure 12, 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 4.11 *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 13 A possible state for Figure 12 is $s_4 = (t_4, x_{41}) = (\texttt{oh0,sink})$. However, this state is not valid, and so for $s_4 = (\texttt{oh0,sink})$, $f_4(s_4) = \infty$. The other two possibilities for s_4 are shown in Figure 12, 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 4.12 *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 4.13 *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:

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

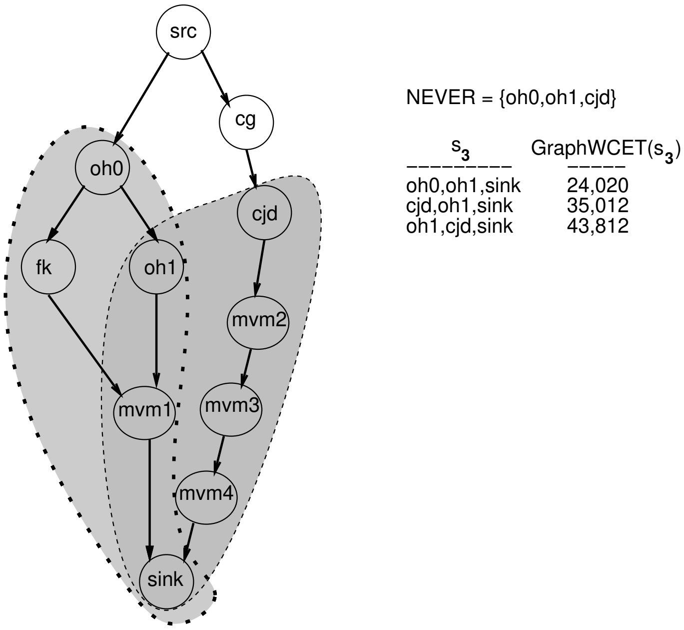


Figure 13: Constructive Heuristic Scheduling Example Stage 3

Definition 4.14 *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 14 Consider Figure 13 where we are in stage 3; the leftmost shaded area shows G_{s_3} for $s_3 = (\texttt{oh0}, \texttt{oh1}, \texttt{sink})$. So we have $t_3 = \texttt{oh0}$ and $x_{32} = (\texttt{oh1}, \texttt{sink})$. One successor of task t_3

is oh1, which is a member of the $NEVER$ set. Thus we use $GraphWCET(\texttt{oh}1, x_{32}) = 21,799$ 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 $\{\texttt{fk}, \texttt{mvm1}, \texttt{sink}\}$ 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 $GraphWCET_{succ}(t_3, x_{32}) = 21,799$. □

Definition 4.15 *Given a state $s_i = (t_i, x_{ik})$, we define the following:*

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

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

Example 15 Consider the leftmost shaded area in Figure 13 again; we have t_3 = oh0, x_{32} = (oh1,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.

4.2.2 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 4.1).

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 $(1, 2, \ldots, n)$ and use index i to refer to current stage.

Thus, since the sink is always schedule in stage n, our approach starts with the second to last stage, stage $n-1$, and progressively works its way back to the first stage, stage 1.

The pseudo-code for the Constructive Heuristic Scheduling Algorithm is shown in Figures 14 and 15. The algorithm of Figure 14 calculates the worst-case execution time for a given stage, whereas the algorithm of Figure 15 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.

The algorithm of Figure 15 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,

$Calc\_WCET\ (G, SWNEVER, i, n, f_{i+1}{}^*, X_i)$ {
  initialize $f_i{}^*$, f_i; $X_{i-1} = \emptyset$;
  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 14: Calculate $WCET$ Algorithm

then clearly t_i cannot be scheduled in stage $n-1$ since no software-task can ever be scheduled after stage $n-1$ (recall that the `sink` is always scheduled last, i.e. 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

$Solve\_order(G, SWNEVER, WCETisr, WCETprsched)$ {
    $\mathbf{n} = |SWNEVER| + 2;$ /* number of stages */
    increase $WCET$ for each task in $SWNEVER$ by $WCETisr + WCETprsched$;
    $\mathbf{f}_n{}^*(s_n) = \mathbf{f_n}(\mathtt{sink}) = 0;$ /* initial values for stage n-1 */
    $X_{n-1} = \{(\mathtt{sink})\};$
    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 */

    $(\mathbf{f_1}^*, X_0) = Calc\_WCET(G, \{(\mathtt{src})\}, 1, n, \mathbf{f_2}^*, X_1);$
    /* 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 $\{(\mathtt{src})\}$ to $Calc_W CET$ */
    $G_{WCET} = \mathbf{f_1}^*(x^*_{01});$ /* annotate G with minimal overall $WCET$ found */
    $G_{task\_order\_list} = x^*_{01};$ /* record the task order found */
}

Figure 15: Constructive Heuristic Scheduling Algorithm

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 4.5) 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 consists 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 *Graph−WCET* 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 consists 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 3: Constructive Heuristic Scheduling Example Stage $n - 1 = 4$

Example 16 Consider Figure 10. We have $n = 5$ stages. For stage 5 we found that $f_5{}^*(\texttt{sink}) = 0$. Table 3 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 12 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 4: Constructive Heuristic Scheduling Example Stage 3

Example 17 Continuing our attempt to schedule Figure 10, we pass now to stage 3. Table 4 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 13 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 5: Constructive Heuristic Scheduling Example Stage 2

Example 18 Next consider stage 2 of the attempt to schedule Figure 10 using the constructive heuristic scheduling algorithm. Table 5 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 optimum. The fact that this algorithm is a heuristic can be verified by applying it to the example of Figure 16.

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)

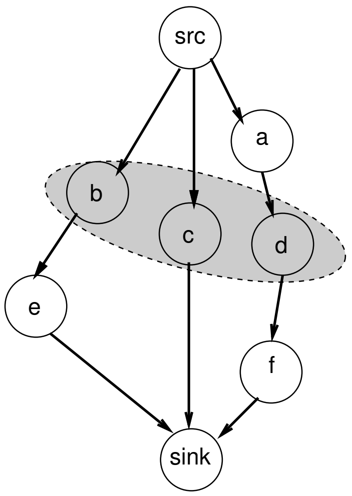
Table 6: Constructive Heuristic Scheduling Example Stage 1

Example 19 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 6 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.

4.2.3 Multiple $NEVER$ Sets of Hardware-Tasks

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.



task	wcet (cycles)
a	5,000
b	3,000
c	20,000
d	15,000
e	5,000
f	11,000

Figure 16: 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.

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 20 We consider a modified version of Figure 10 where the four tasks `mvm1-4` are all executed on the same hardware module `mvm`. Figure 17 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 17 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 5. except that it would have seven by seven entries, one row/column per task in a $NEVER$ set.

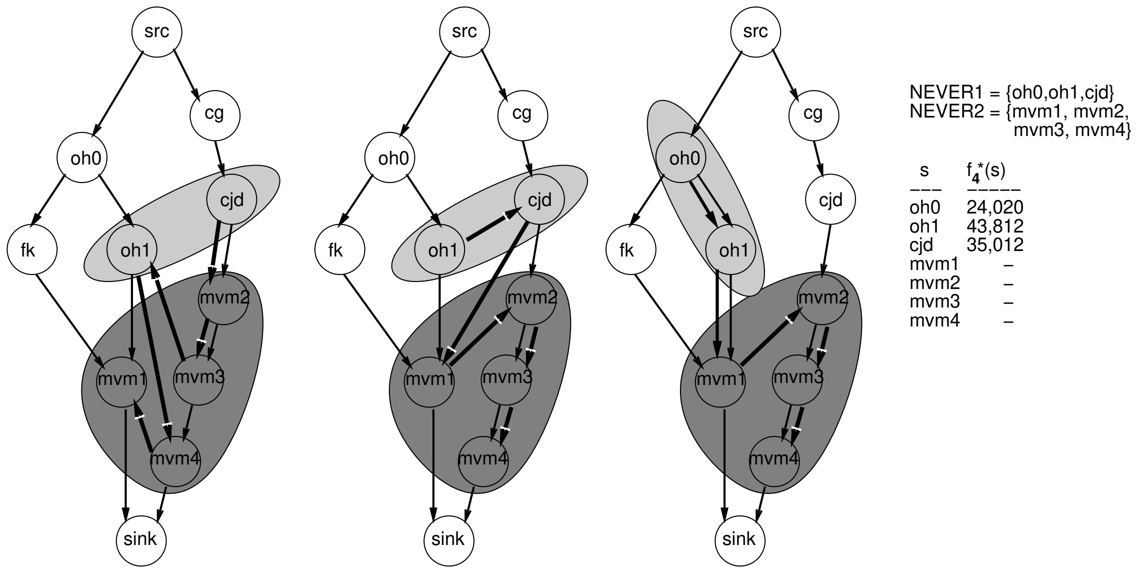


Figure 17: Multiple $NEVER$ Set Example

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 14 and 15. The only difference in calling algorithm *Solve_order* of Figure 15 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.

Figure 18 shows the modifications necessary to convert Figure 15 to handle multiple $NEVER$ sets of hardware-tasks. Note that the only changes are in lines 1, 2, and 7 of Figure 18. The final task order found in x^*_{01} contains all tasks in any $NEVER_i \in NEVERSETS$. Thus, the task order for tasks in the same $NEVER$ set can be extracted from x^*_{01} by simply removing the relevant tasks from x^*_{01} in the order they are found.

$Solve\_order(G, NEVERSETS, WCETisr, WCETprsched)$ {

$SWNEVER$ = first set in $NEVERSETS$;

$\mathbf{n} = (\sum_{NEVER_i \in NEVERSETS} |NEVER_i|) + 2$; /* number of stages */

increase $WCET$ for each task in $SWNEVER$ by $WCETisr + WCETprsched$;

$\mathbf{f_n}^*(s_n) = \mathbf{f_n}(\texttt{sink}) = 0$; /* initial values for stage n-1 */

$X_{n-1} = \{(\texttt{sink})\}$;

for $(i = n-1; i > 1; i--)\{$ /* go through the stages in reverse order */

$(\mathbf{f_i}^*, X_{i-1}) = Calc\_WCET(G, NEVERSETS, i, n, \mathbf{f_{i+1}}^*, X_i)$;

/* 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 18: Constructive Heuristic Scheduling Algorithm with Multiple $NEVER$ Sets

Example 21 Consider a Figure 17 again. Let x^*_{21}, x^*_{22} and x^*_{21} denote the decision variables for the left, middle and right-hand graphs shown in Figure 17, respectively. For the leftmost DAG, we have x^*_{21} = {`cjd, mvm2, mvm3, oh1, mvm4, mvm1, sink`} from which we would extract order {`cjd,oh1`} for $NEVER1$ and order {`mvm2, mvm3, mvm4, mvm1`} for $NEVER2$. For the DAG in the middle, we have x^*_{22} = {`oh1, cjd, mvm1, mvm2, mvm3, mvm4, sink`} from which we would extract order {`oh1,cjd`} for $NEVER1$ and order {`mvm1, mvm2, mvm3, mvm4`} for $NEVER2$. Finally, for the rightmost DAG, we have x^*_{23} = {`oh0, oh1, mvm1, mvm2, mvm3, mvm4, sink`} from which we would extract order {`oh0,oh1`} for $NEVER1$ and order {`mvm1, mvm2, mvm3, mvm4`} for $NEVER2$.

It turn out that the final optimal solution is found from leftmost DAG and yields x^*_{01} =

{`src, oh0, cjd, mvm2, mvm3, oh1, mvm4, mvm1, sink`} from which we would extract order {`oh0, cjd,oh1`} for $NEVER1$ and order {`mvm2, mvm3, mvm4, mvm1`} for $NEVER2$.
□

The changes needed to alter Figure 14 to handle an input of multiset $NEVERSETS$ instead of the single set $SWNEVER$ are so few that we will simply describe them here in words. The first change is to define the j^{th} task of $NEVERSETS$ to be the j^{th} task encountered when processing each $NEVER_i \in NEVERSETS$ one by one in the same order each time (i.e. in the order they are stored in $NEVERSETS$). The second change from the single $NEVER$ set algorithm shown in Figure 14 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 4.15, where d denotes the number of $NEVER$ sets contained in the multiset $NEVERSETS$.

4.2.4 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 $((|V_{s_i}| + |A_{s_i}|))$. 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 $(|V_{s_i}|+|A_{s_i}|)$ for calculating f_i, we end up with an asymptotic upper bound of $O((n-2)^2 * (|V_{s_i}| + |A_{s_i}|)) = O((n^2) * (|V| + |A|))$ calculations for one stage (i.e. for $Calc\_WCET$ of Figure 14).

For $Solve\_order$ shown in Figure 15 or Figure 18, none of the lines take time greater than $O(n^2 * (|V| + |A|))$. Thus, since we call $Calc\_WCET$ at most n times, our constructive heuristic scheduling algorithm with multiple $NEVER$ sets takes time $O(n^3 * (|V| + |A|))$. Assuming we can bound V and A by constants, we have a polynomial-time algorithm.

4.2.5 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 R4000. 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 R4000 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 R4000 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 4.3), 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 22 Consider Figure 10. We use the priorities found in Example 19. 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[MWWL96, LM95]). Notice how the critical path runs through both hardware and software in different execution paths. Table 7 shows the calculation. The overall *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 4.5). 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 4.3). 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)

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	4,264		
oh1	4,400	mvm3	4,400
oh1	4,400	mvm4	4,400
oh1	4,335	mvm1	4,400

Table 7: *WCET* Calculation Example

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.

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.

4.3 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 4.3 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.

4.3.1 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 4.2.

[1] For example, the major result of [HWS95] was a 66% reduction in context switch cost.

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 4.16 *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.*

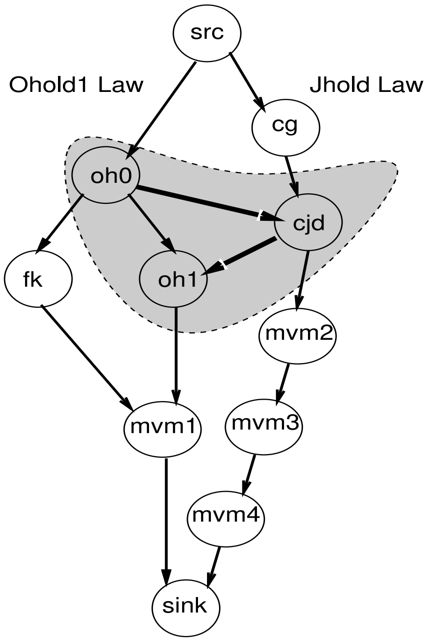
Presumably we found Π using the constructive heuristic scheduling algorithm of the previous section.

Definition 4.17 *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 4.18 *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.



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 19: DAG, $WCET$ and Π Example

Definition 4.19 *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 4.20 *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 23 Consider the DAG shown in Figure 19 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 emboldened edges in Figure 19. 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 20, Figure 21,

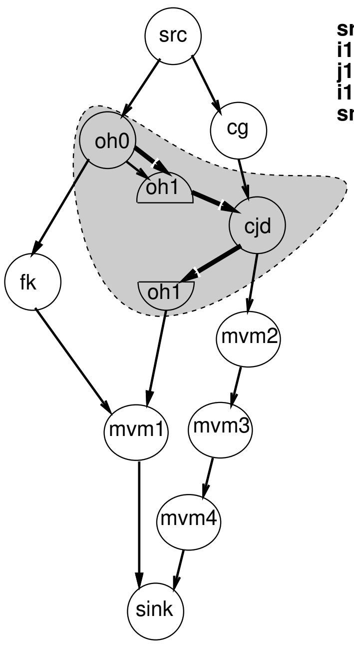


Figure 20: DAG With Out-of-order Execution Example

Example 24 and the subsequent proofs, the abbreviation **p** stands for a call to the `priority_scheduler` code, **sc** stands for a call to the `save_context` code and **rc** stands for a call to the `restore_context` code.

Example 24 Figure 20 shows a graphical representation of the execution of the DAG of Figure 19 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 $i_1 = \texttt{oh1}$ 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. □

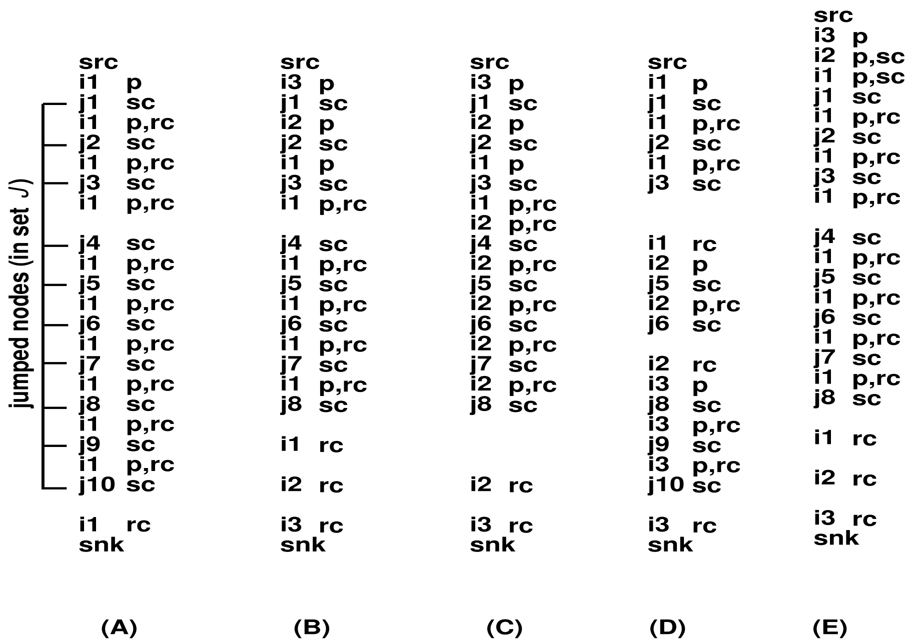


Figure 21: Extra Priority Scheduler and Context Switch Time Examples

Example 25 Let's consider the three examples of Figure 21. 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 $(\mathbf{A})$, 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 $\mathbf{p}$ and to $\mathbf{rc}$ before actually running any instructions of i_1 itself. Then, when a task in J is ready to execute, a call to $\mathbf{sc}$ has to be made since i_1 is not finished yet. Notice that no $\mathbf{rc}$ calls are needed for any of the tasks in J since each $j \in J$ runs to completion. $\Box$

Next we propose two theorems about the number of additional calls to $\mathbf{p}$, $\mathbf{rc}$ and $\mathbf{sc}$ 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 4.2 *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 $I = \{i_1\}$ *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*

$$|J| * (\mathbf{p} + \mathbf{sc} + \mathbf{rc}). \tag{4.1}$$

Proof: In the worst case i_1 executes before j_1, causing an extra call to $\mathbf{p}$, but does not finish execution. Next j_1 becomes ready to execute, causing a call to $\mathbf{p}$ and $\mathbf{sc}$. Since the call to $\mathbf{p}$ would have happened anyway, only the $\mathbf{sc}$ call is additional. After j_1 finishes, in the worst case there is exactly enough time for only a single extra call to the $\mathbf{p}$ and to $\mathbf{rc}$ 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 $\mathbf{sc}$ 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 **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. $\sharp$

Example 26 An example of the worst case scenario is shown in **(A)** of Figure 21, which shows the case for $q = 10$. The total number of extra calls is $10 * (\mathbf{p} + \mathbf{rc} + \mathbf{sc})$. $\Box$

Theorem 4.3 *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*

$$(|(J - (J \cap I)) \cup I| - 1) * (\mathbf{p} + \mathbf{sc} + \mathbf{rc}). \tag{4.2}$$

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 4.2, 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 **p**, **rc** and **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 4.2. 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 21, **(E)**, for an example). This causes an extra **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 **p** and **sc** are incurred. Similarly, i_{k-1} becomes ready, incurring yet another **p** and **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 **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 **p** for i_1 will have been extra, and only this call to **p** now that $j_{q_{k+1}}$ is done will not be an extra call – but the call to **rc** will be additional (see Figure 21, **(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 **rc** will occur (the remaining $|I_{k+1}| - 1$ calls to **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 21, **(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. $\sharp$

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 4.2, 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` and `priority_scheduler`.

4.3.2 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 [LM95]. 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[MWWL96, LM95] 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:

$$WCET\_reload\_icache = (\lceil \frac{binarycodesize - 1}{icachelinesize} \rceil + 1) * (\text{time to load a single icache line}) \tag{4.3}$$

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 27 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 R4000 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 4.3:

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

□

Practical Considerations in Instruction Cache Analysis

As in Section 4.2.5, we consider the case where the processor is a MIPS R4000. Note that the MIPS R4000 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 4.2.5.

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.

4.3.3 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 4.2 and its corresponding Equation 4.1, 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:

$$|J| * (WCETprsched + WCETsavecntxt + WCETrestorecntxt + WCET\_reload\_icache) \tag{4.4}$$

Similarly, by Theorem 4.3 and Equation 4.2, for a given G and I with associated J, an upper bound on the increase in *WCET* for G is given by the following:

$$(|(J-(J\cap I))\cup I|-1)*(WCETprsched+WCETsavecntxt+WCETrestorecntxt+WCET\_reload\_icache) \tag{4.5}$$

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

4.3.4 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 4.3 and thus software-tasks are not all necessarily atomic.

We first compute the priorities by the algorithm of Section 4.2 for multiple $NEVER$ sets. Thus, we have an order of software- and hardware-tasks contained in $NEVER-SETS$ 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 4.3 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 4.5 of the previous section: if the amount of space (unused cycles) is greater than or equal to Equation 4.5, then yes, otherwise no. That is the

$Execute\_out\_of\_order(G, \Pi, NEVERSETS,$
$WCETprsched, WCETsavecntxt, WCETrestorecntxt)$ {
$SWNEVER$ = 1st set in $NEVERSETS$; $m = |SWNEVER|$;
/* Get set and number of software-tasks */
$\Omega = (\texttt{src}, p_1, p_2, \ldots, p_m)$ where $p_i \in SWEVER, 1 \leq i \leq m$,
and $\Pi(p_1) > \Pi(p_2) > \ldots > \Pi(p_m)$;
/* Ω stores the source followed by the software-tasks in priority order */
$p_0 = \texttt{src}$; /* now we have $\Omega = (p_0, p_1, p_2, \ldots, p_m)$ */
W = $WCETprsched + WCETsavecntxt + WCETrestorecntxt$;
$I = \emptyset$; $J = \emptyset$; $\Psi = \emptyset$; /* Ψ keeps track of new precedence constraints */
$i = 1$; $WCET\_reload\_icache = 0$; /* i counts the number of tasks in set I plus one */
/* the following **for** loop considers allowing sw-tasks $(p_m, p_{m-1}, \ldots, p_2)$ to execute early */
for $(l = m;\ l \geq 2;\ l--)$ {
  $k2 = p_{l-1}$;
  if $(k2 \in J)$ $v = 0$;
  else $v = 1$; /* v counts the number of tasks skipped by p_l and not already $\in J$ */
  $new\_prec\_task = k2$;
  for $(k1 = p_{l-2}$ to $k1 = p_0)$ {
    if ($\exists$ a precedence constraint $\{k2 \rightarrow p_l\}$)
      continue; /* exit inner **for**(k1=...) loop */
    if ($\texttt{get\_space}(k1, k2) \geq$
      $(i + v + \texttt{num\_tasks\_skipped}(\Omega)$ - 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 $(k2 \neg\in J)$ $v++$;
  }
  if (new_prec_task) {
    $\Psi = \Psi \cup \{new\_prec\_task \rightarrow p_l\}$;
    /* add new precedence constraint $\{new\_prec\_task \rightarrow p_l\}$ to Ψ */
    update I, J;
    reduce $\texttt{get\_space}(new\_prec\_task, after\_prec\_task)$
      by $(i + \texttt{num\_tasks\_skipped}(\Omega))$*(W + $WCET\_reload\_icache_i$);
    $i++$;
  } **else** { $\Psi = \Psi \cup \{p_{l-1} \rightarrow p_l\}$; } /* add consecutive precedence constraint to Ψ */
}
return(Ψ, $WCET\_reload\_icache$);
}

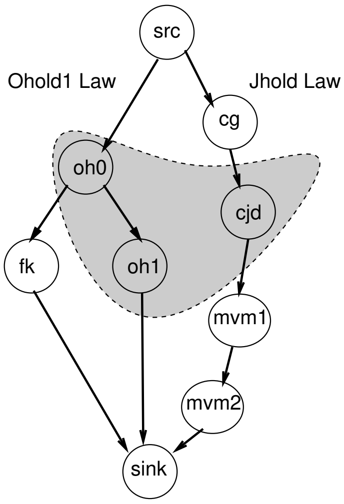
Figure 22: *Execute Out-of-order* Algorithm

insight behind the heuristic *Execute Out-of-order* procedure of Figure 22.

We describe now the heuristic algorithm of Figure 22 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 4.2.3, 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 4.5. Note that we calculate Equation 4.5 in Figure 22 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; otherwise we add the precedence constraint of strict in-order consecutive execution, namely $\{p_{l-1} \rightarrow p_l\}$. 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 algorithm 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.

Example 28 **[Sample Application of** ***Execute Out-of-order*** **algorithm]** Consider Figure 23, which shows the $BCET$ and $WCET$ for each task, the icache refill $WCET$ for the



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 23: Example With $WCET$ Calculation of Instruction Cache Refill Time

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 4.2.5, from which we also find that $W = 422$). At this point in the algorithm of Figure 22, 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 22) 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 22 for the example of Figure 23.

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`. □

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 4.2.5, we will consider the specific case where the processor is a MIPS R4000. We use the following costs, obtained by analyzing our run-time scheduler software code executed on a MIPS R4000 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 22, 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 4.2.5. 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 4.3 in Section 4.3.3, namely Equation 4.5, 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	90	mvm3	4,400
oh1	4,400	mvm4	4,400
oh1	4,402	mvm1	4,400

Table 8: *WCET* Calculation Example

Example 29 [WCET calculation] Consider Figure 23. 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 28 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 22 gives us $WCET\_reload\_icache$ = 4050. Using our costs for $WCETprsched, WCETsavecntxt$, $WCETrestorecntxt$ and $WCET\_reload\_icache$, we find that $W = 422$. From Equation 4.5, we find that

$(|(J - (J \cap I)) \cup I| - 1) * (WCETprsched + WCETsavecntxt + WCETrestorecntxt + WCET\_reload\_icache) = 1 * (422 + 4050) = 4472.$

Table 8 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.

4.4 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 4.3, one might be tempted to go back to the original formulation of the Constructive Heuristic Scheduling Algorithm of Section 4.2 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 4.3? It turns out that there are. Consider the following example:

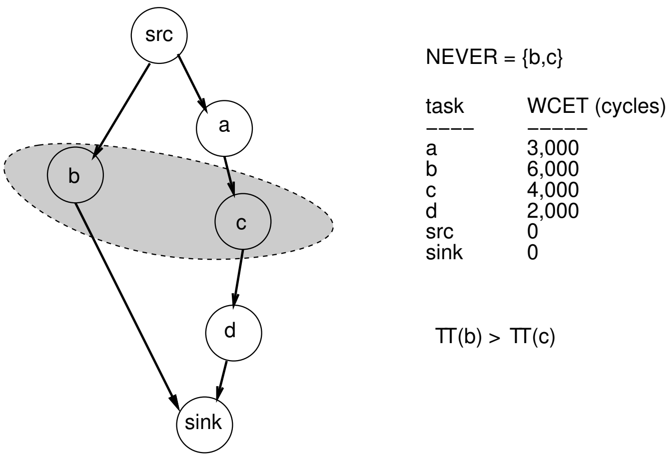


Figure 24: Constructive Heuristic Scheduling Example of Suboptimal Result

Example 30 Consider Figure 24. 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 4.3.4 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. □

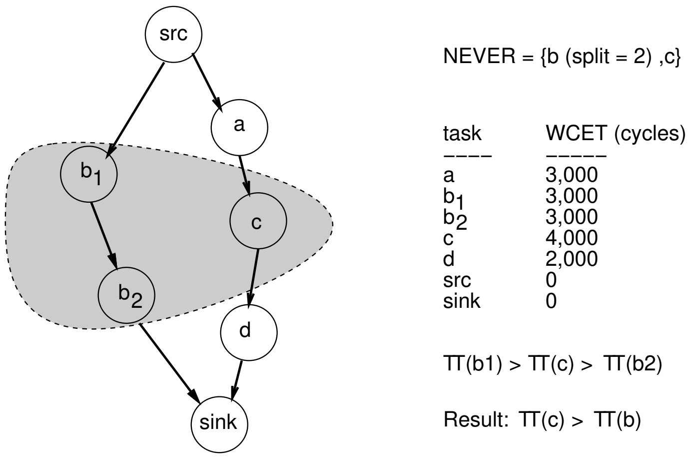


Figure 25: Example of Scheduling with Task Splitting

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 31 Consider Figure 25. 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 25. 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 4.3.4 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. □

4.5 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.

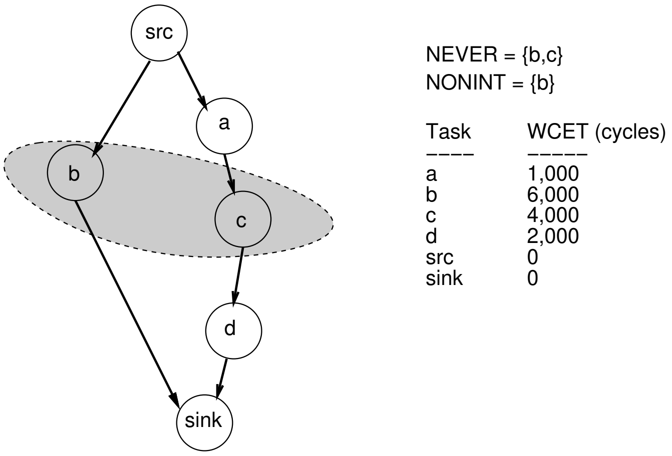


Figure 26: Example Specification of Noninterruptible Task

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 4.2 are modified to take into account that these processes are noninterruptible by simply retaining Assumption 4.3, 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 *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 iteration on the critical region or semaphore allowed is that which occurs in each execution of the DAG. Loops defined on a critical region are not allowed since a DAG cannot contain loops and remain acyclic. However, loops without critical regions *are* allowed in individual C and Verilog HDL 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 32 In Figure 26 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. □

4.6 Summary

In this chapter we showed how to efficiently solve for the order of tasks in the same NEVER set using a heuristic that constructively builds a solution from the DAG representing the partial order needed for the hardware- and software-tasks to execute. Then we extended the heuristic to handle the case of multiple NEVER sets in hardware.

We next considered the case where software-tasks in the same NEVER set are allowed to interrupt and preempt each other. To handle this case, we statically calculate an upper bound on the number of extra calls that may be necessary to our kernel code – namely, to the priority scheduler and context switch code. Using this number, we find a total upper bound on the WCET with specified software-tasks allowed to preempt other particular software-tasks. Based on this analysis, we propose a greedy heuristic for allowing some lower priority tasks to execute during idle time, thus possibly being preempted by higher priority tasks.

Finally, we presented a task-splitting method to improve our results in some cases where our assumptions change, as well as a feature to declare certain tasks to be noninterruptible, thus providing support for critical regions.

Chapter 5

Implementation and Experimental Results

This chapter consists of three sections. In the first section, we present an overview of the tool flow implementing the design approach and algorithms of the previous chapters. In the second, we present an example of a PUMA robot control to show how a design can be successfully synthesized using the CAD system described. We verify the synthesis results of the PUMA controller via simulation. Finally, in the last section we present a sample prototype implementing the split run-time scheduler in a Haptic robot. The Haptic robot prototype was achieved by modifying existing Haptic robot in the Computer Science Robotics laboratory at Stanford.

5.1 Design System Implementation

Figure 27 (repeated from Figure 6 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 HDL and software tasks are written in C. Constraints include relative timing constraints, a single rate constraint, and resource constraints in the form of

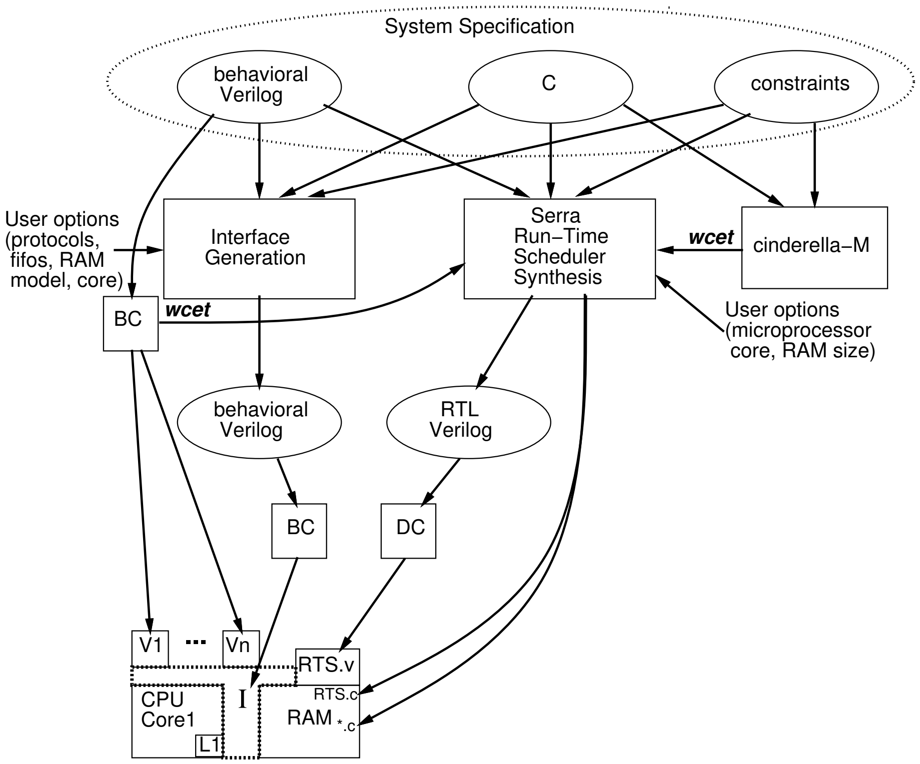


Figure 27: Tool Flow and Target Architecture

$NEVER$ sets. Precedence constraints are implicit in the task specification which takes the form of a Directed Acyclic Graph. SERRA performs run-time scheduler synthesis and *worst-case execution time (WCET)* analysis. Satisfaction of relative timing constraints (minimum and maximum separation) in hardware blocks is dealt with in hardware control synthesized by the THALIA2 tool. THALIA2 generates a hardware FSM implementing a CFE specification of the system with relative timing constraints (CFEs and THALIA2 were described briefly in Section 2.4 [CM96, Coe96, CM97]).

The system-level tasks, written Verilog HDL 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. Worst-case software execution time is found by the tool CINDERELLA-M, which was described in Section 2.5 and which takes as input C programs for the software tasks and outputs a *WCET* for each software task (note that bounds on loops must be provided by the user) [MWWL96, LM95]. 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).

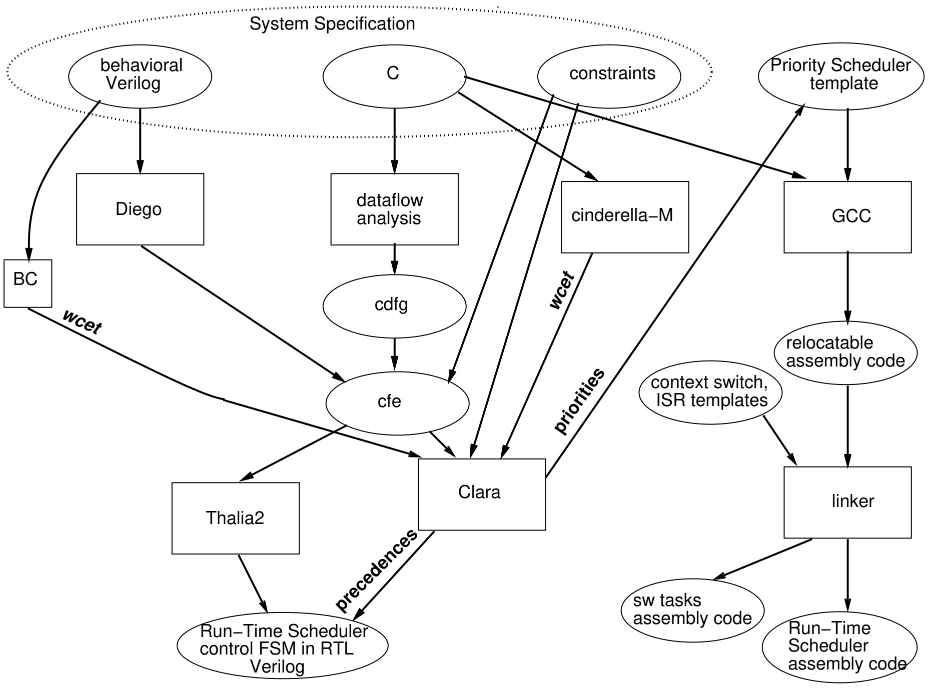


Figure 28: Block diagram of Serra: the boxes indicate tools and the ovals indicate data.

5.1.1 Serra Run-Time Scheduler Analysis and Synthesis

The Serra design tool is shown in Figure 28, which expands the box labelled Serra in Figure 27. 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; from it we extract a CFE describing the task flow of the system. Diego can extract a

CFE description from a task written in Verilog HDL; for example, given the main task in Verilog HDL, DIEGO can generate in CFE format the sequence of task invocations (calls) from the main task. A *WCET* for each task is calculated using BC^{TM} or CINDERELLA-M; next, the CFEs are annotated with the *WCET* calculated for each hardware or software task. These *WCET* values are used to annotate the leaf tasks in the final DAG of the system specification. Figure 10 showed a sample DAG and a corresponding table with the *WCET* annotations. Finally, 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 HDL [CM96, MCSM96]. The timing, resource and precedence constraints specified in the CFEs input to THALIA2 are translated into a finite-state machine implementation if a solution is found which satisfies the constraints.

The constructive heuristic scheduling algorithm is implemented by CLARA, which generates the static priorities for the software and hardware tasks. Since we assume that all hardware-tasks are noninterruptible, in the case of hardware-tasks in the same $NEVER$ set, the static priorities found by CLARA are converted into precedence constraints enforcing the order indicated. 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 4.4), and *noninterruptible* software-tasks (Section 4.5). Furthermore, CLARA can generate precedence constraints among software-tasks in a single $NEVER$ set 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 4.3.4. CLARA has been implemented in 15,000 lines of C.

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 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 an order different from that found by the heuristic of Section 4.3.4. 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 and tuning of the run-time system, as well as synthesis for each particular implementation.

5.2 Design Case Study: PUMA Robot Arm

In this section, we use and show the Serra system to design a robot controller for manipulating two PUMA arms, which are a standard in the robotics industry [AKB86, Uni84] (Figure 4 showed two PUMA arms grasping an object). The controller implements concurrent models of two "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, where by dynamic we mean that the exact times resources are allocated is not statically determined but instead is determined at run-time. We describe how we simulated the final implementation.

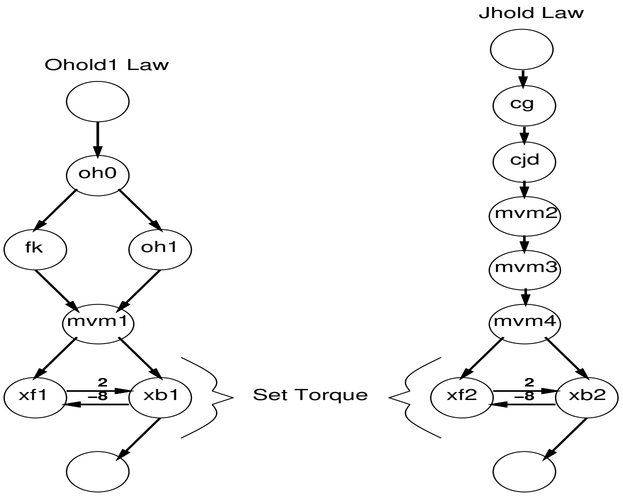
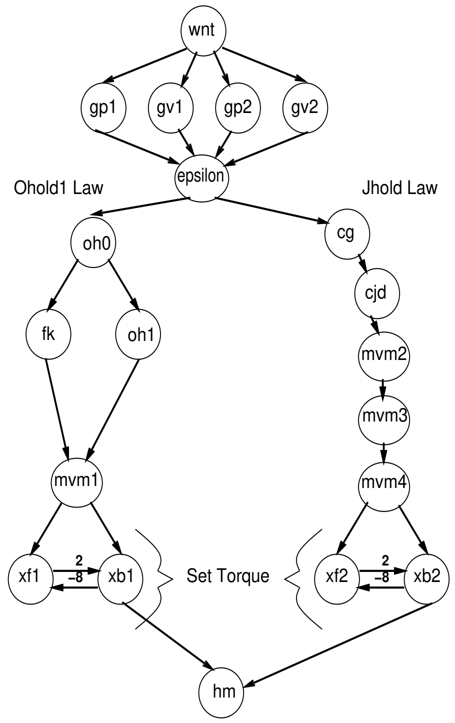


Figure 29: Directed Acyclic Graphs of Ohold1 Law, Set Torque, and Jhold Law with Relative Timing Constraints

5.2.1 Two PUMA Arms

For our example, we consider the robot control algorithm of Figures 8 and 9. 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 29; the full DAG is shown in Figure 30. 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 27 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. 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



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

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

Figure 30: DAG of Robot Arm Controller with Relative Timing Constraints

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 28, 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 29, we calculate a *WCET* for of 42,113 for Figure 29 with out-of-order execution. This provides for the upper bound on execution speed for the tasks in Figure 29 under worst-case conditions.

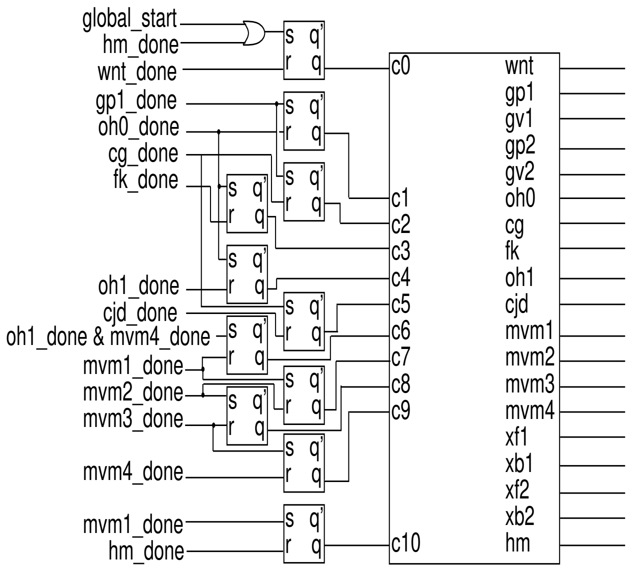


Figure 31: Final Hardware Portion of Run-Time Scheduler

Figure 31 shows the hardware portion of the run-time scheduler. Signals `wnt`, `gp1`,`gv1`,..., `hm` in Figure 31 are the *start* events for the corresponding tasks in Figures 29 and 30. 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 [CM96, MSM97]. Note that Figure 31 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` $\rightarrow$ `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 8 and pictured graphically in Figure 20. 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 HDL 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`.

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

The complete, flattened DAG with relative timing constraints is shown in Figure 30 (reprinted from Figure 9 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 30 but not in Figure 10 – `wnt`, `gp1`, `gv1`, `gp2`, `gv2`, `xf1`, `xb1`, `xf2`, `xb2`, `hm` – together take 57,200 cycles in the worst case. Since our MIPS R4000 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.

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

Table 10: Results for the synthesis of hw-tasks.

Table 9 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 10, we see the results for the synthesis of the hardware tasks of Figure 10 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 10 shows the number of gate equivalents of hardware required using the LSI 10K Logic library. The *WCET* values are initial estimates based on the number of control steps needed to execute the hardware as scheduled by the Behavioral Compiler^{TM}. The estimates are rounded up and then scaled to allow direct comparison with the MIPS R4000 processor operating at 100MHz (we clock the hardware at 10 MHz).

5.2.2 Verilog Simulation

Using a MIPS R4000 processor core model in Verilog HDL, we simulated the Robot Arm Controller, with its synthesized run-time scheduler, in Verilog HDL using Synopsys' Chronologic VCS^{TM}. The simulation utilized memory-mapped I/O as the medium of interface between the hardware and software tasks involved in the robotics control algorithm.

The run-time scheduler comprises, for hardware, the Run-Time Scheduler module and its associated set-reset latches as shown in Figure 31, and, for software, the Priority Scheduler and the Interrupt Service Routine.

Figure 32 shows a simplified block diagram of the Verilog simulation showing the memory mapping for the Priority Scheduler and Interrupt Service Routine, the three robot control software algorithms, and the memory-mapped and local *start* and *done* vectors. Note that the diagram neglects to depict the interrupt signal and the existence of the L1 cache. Our target architecture consists of a MIPS R4000 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.

The Priority Scheduler (PRS), along with the robot control software algorithms are automatically placed in main memory starting at word address 0x64 by the linker. The Interrupt Service Routine (ISR) is placed in memory at location 0x2000_0060. This location was chosen because the Program Counter (PC) jumps to 0x2000_0060

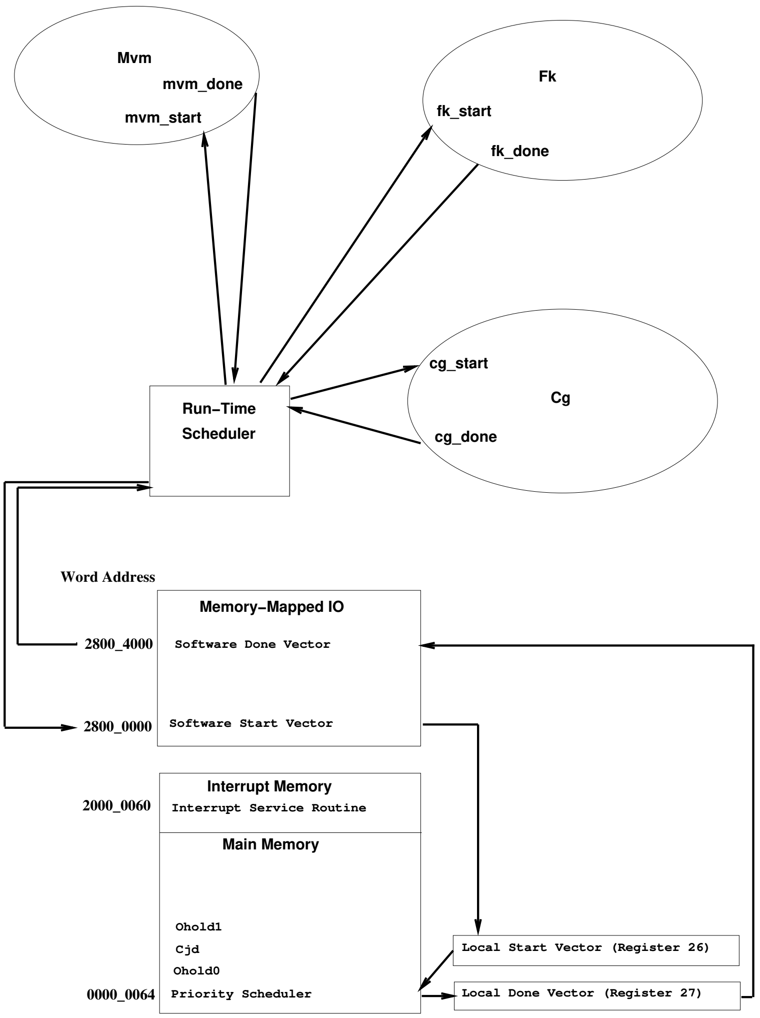


Figure 32: Simplified Block Diagram of the Simulation

whenever an interrupt is asserted on the CPU.

The memory-mapped *start* and *done* vectors were chosen to be at 0x2800_000 and 0x2800_4000, respectively. The lower 32 bit contents of 0x2800_0000 are loaded into register r26 by the ISR whenever an interrupt is asserted, whereas the contents of register r27 are written out to 0x2800_4000 by the PRS once a specific software task is done. Registers r26 and r27 were chosen because they are kernel-reserved registers.

5.2.3 Run-Time Scheduler Software

Compiling the Priority Scheduler

The Priority-Scheduler code is written in C in file *prs.c*. The task of this code is to:

- Look at the on-chip (32-bit) start register (register 26)
- Determine which corresponding software task should be run
- Write out the proper value to the on-chip done vector (register 27) once a specific software task is done. The contents of register 27 are later transferred to the done vector in memory-mapped I/O.

However, reading from, and writing to, an on-chip register is somewhat difficult to do in C. To get around this problem, we can either inline assembly code in the C code, or change the compiled C. We have chosen the latter option since it was trivial to change the assembly code by hand.

Thus the procedure for compiling *prs.c* is:

- Compile *prs.c* with the standard *cc* compiler on the SGI.
- Edit the resultant file, *prs.s*, from the previous step to have all instances of start and done access registers 26 and 27 respectively. For instance, the code fragment

  start = start & ˜CJD;

  in prs.c compiles to

  .loc 2 178

  start = start & ˜CJD;

  lw $12, $36($29) // $29 is the stack pointer

  and $13, $12, -3

sw \$13, \$36(\$29)

in *prs.s*. This needs to be be modified to access register 26, where the start value really resides, instead of the stack, like so:

.loc 2 178

start = start & ~CJD;

add \$12, \$0, \$26

and \$13, \$12, -3

add \$26, \$0, \$13

- Also change

  done_ioif = done;

  lw \$12, \$32(\$29)

  sw \$12, \$40(\$29)

  to

  done_ioif = done;

  add \$12, \$0, \$27 // This transfers the done vector to register 27

  sw \$12, 0xA0010000 // \$2800_4000 in memory-mapped I/O

  Note that just the instruction *sw \$27, 0xA001000* will not work as the compiler has allocated space for two instructions during the compilation of *prs.c* (although inserting a *nop* would work).

- Assemble *prs.s* with the MIPS assembler available on the SGI using *cc*

- Copy the objectcode, *prs.h*, to *objectfile.h*, and *prs.start* to *objectfile.start*. During initialization of the simulation, *memory.v* will read in *objectfile.h* and place it in main memory at location 0x64 (it actually varies depending on the start location specified by the linker in $prs.h$).

- For purposes of verification, the contents of main memory just after initialization can be viewed in the file *memstart.txt*.

Interrupt Service Routine

The Interupt Service Routine code, *isr.s*, is written in MIPS assembly and transfers the start vector in memory-mapped I/O to register 26 when an interrupt is asserted by the hardware Run-Time Scheduler ($rts.v$); isr.s is assembled with the *cc* assembler on an SGI Indigo.Because the MIPS R4000 CPU transfers program execution to 0x2000_0060 whenever its interrupt pin is asserted, we have to forcibly place the ISR code there. This is simply done by changing the first line of *isr.h* to read 2000_0060. However this means that we have to ensure *isr.s* does not contain any "hard-coded" jumps/branches to code within itself.

Initialization

Registers 26 and 27 have to be initialized to zero at the start of the simulation, otherwise a software task may be prematurely started. This is done in $regfile.v$.

At simulation initialization, *memory.v* will read in *isr.h* and place it at 0x2000_0060. This can be verified by looking at the contents of the file *memstart_int.txt*.

5.2.4 Run-Time Scheduler Hardware

The Run-Time Scheduler hardware modules *rts.v* and *srlatch.v* are the heart of the run-time scheduler system. The file *rts.v* handles resetting the hardware modules involved in the control flow, passes the start and done vectors between each of the hardware and software tasks, and restarts the control flow after each iteration. The file *srlatch.v* models a normal set-reset latch.

Initialization

The initial block present in rts.v asserts a RESET on the following hardware modules: *MVM_3.v_rst* (the hardware module for task `mvm`), *CG_2.v* (task `cg`), *FJ_2.v* (task `fj`), and *robotarmcheckzero.ver*2, which was synthesized with THALIA2, at simulation start time. It also starts off the contol flow by setting `c0` high for 10 processor cycles.

Control flow module

The synthesized Verilog HDL module of the control flow implemented in the simulation is stored in file *robotarmcheckzero.ver*2. It has inputs `c1` to `c6` with the following correspondences: `c1-oh0`, `c2-mvm`, `c3-fk`, `c4-oh1`, `c5-cjd`, `c6-cg`. It also has outputs `wnt`, `gp1`, `gv1`, `gp2`, `gv2`, `mvm`, `cjd`, `cg`, `oh0`, `oh1`, `fk`, `xf1`, `xb1`, `xf2`, `xb2` which are actually the start vectors for each similarly-named task as shown in Figure 31.

Start/done for hardware tasks

File *rts.v* instantiates a set-reset latch for each hardware task. Thus, the hardware *start* and *done* vectors are not in registers but are hooked up directly to each harware task. For example, the task `cg` has a set-reset latch called *trigger_cg*. When task `cjd` completes, it sends a *done* signal to the set (S) port of *trigger_cg*, and *trigger_cg* then outputs a high. This output is hooked up to the input pin `c6` of *robotarmcheckzero.ver*2. Hence a cycle later *robotarmcheckzero.ver*2 will assert a high on its output pin `cg`. Signal `cg` feeds directly into *CG_2.v*, so this starts off task `cg`. Once task `cg` is done, *CG_2.v* sends out a *done* signal to the reset

(R) port of *trigger_cg*, causing *trigger_cg's* output to go low; this then prompts *robotarmcheckzero.ver2* to start the next task, which in this case is `xf2`.

Start/done for software tasks

File *rts.v* also instantiates a set-reset latch for each software task. For software tasks, however, a start *value* has to be sent to the start vector in memory-mapped I/O. This is accomplished by feeding in the output (*start*) pins corresponding to each software task from *robotarmcheckzero.ver2* into a register and concatenating them thus *HW_data_in_in* = {*61'b0, oh1, cjd, oh0*} . *HW_data_in_in* was chosen as a 64-bit register because data transfers to and from memory-mapped I/O occur in double-words. The initiation of a software task is detected whenever any of the pins `oh1`, `cjd`, or `oh0` go from low to high. When this happens, a series of steps follow:

- *rts.v* sends a write request to memory-mapped I/O.
- *rts.v* sends a number indicating the location where it wants to write to. This number is an index offset, with each index step being a double-word, from the start of memory-mapped I/O (word address 0x2800_0000). Normally we want to send a 0 to have the memory-mapped start vector be at 0x2800_0000.
- *rts.v* then waits for an OK from memory-mapped I/O (it looks at the pin hw_ok).
- When the OK arrives, *rts.v* asserts an interrupt on the CPU (*IntB = 1'b0*), writes out the contents of *HW_data_in_in* to 0x2800_0000 and deasserts the write request.
- A cycle later, the interrupt is deasserted to prevent the CPU from thinking there were two interrupts requested instead of just one.

A few cycles after the completion of a particular software task, another series of steps, which also involve the Priority Scheduler, follow:

- The on-chip done vector is written to its corresponding memory-mapped I/O location by the Priority Scheduler. The on-chip done vector is actually written to the L1 cache first and then, because the R4000 implements a write-through cache, to memory-mapped I/O.
- Whenever a cache line is written to memory, *L1cache.v* sets *PadAddrValid* high
- *memory_mapped_io.v* notices that *PadAddrValid* is high and checks to see whether the cache line is being sent to memory-mapped I/O.
- If it is, *memory_mapped_io.v* checks whether the address written to is where the memory-mapped done vector resides (0x2800_4000).
- If so, a cycle later it transfers 64-bits starting at 0x2800_4000 to a register called *done* in *rts.v*.
- *rts.v* looks at the bits of *done* and asserts a *done* signal corresponding to which bit is on. For instance, if *done*[1] is on, *rts.v* asserts *cjd_done* for two cycles and feeds it to the set port of *trigger_cg* and *trigger_oh*1, thereby starting those tasks. It also sends it to the reset port of *trigger_cjd*, thus causing *robotarmcheckzero.ver*2 to deassert its ouput pin `cjd`, indicating that task `cjd` is finished.

Restarting the control flow

At the end of the last task in the control flow, we would like task execution to restart at `wnt`. This is achieved by asserting `c0` for a cycle at the completion of the last task. We know what the last task is and the detection of the *negative edge* of a signal from the last task's output pin from *robotarmcheckzero.ver*2 is defined as its completion.

5.2.5 Running the Simulation

We start the simulation by executing Chronologic VCS^{TM} with file *FBLD.cmd*, which contains a list of the Verilog HDL files that we want Chronologic VCS^{TM} to compile. Some highlights from the simulation are outlined below. Note that trace commands in Chronologic VCS^{TM}, such as VCDPLUSON and VCDPLUSTRACEON in *system.v*, should be turned off as they tend to significantly slow down the simulation by periodically writing to disk.

Assertion of an Interrupt

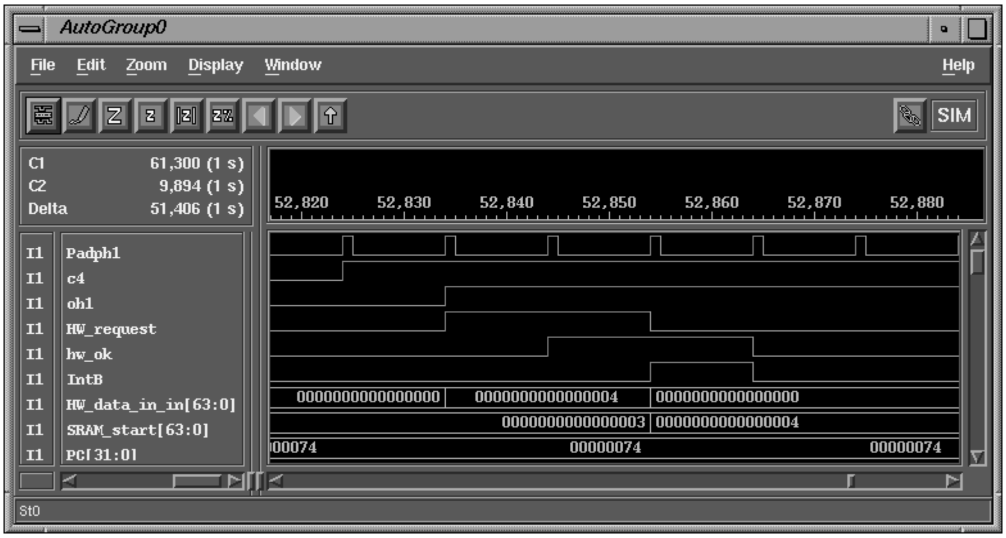


Figure 33: Interrupt Asserted

In Figure 33 we see the process by which the Run-Time Scheduler hardware signals the start of a software task. We see `c4` going high followed a cycle later by `oh1`'s start vector being asserted. This prompts *rts.v* to request a write to memory-mapped I/O (*HW_request*), with the value that is going to be written out residing in *HW_data_in_in* (the value four) . Memory-mapped I/O replies with *hw_ok*. Then *rts.v* asserts an interrupt on the CPU and writes the value four from *HW_data_in_in* to the start vector in memory-mapped I/O (*SRAM_start*).

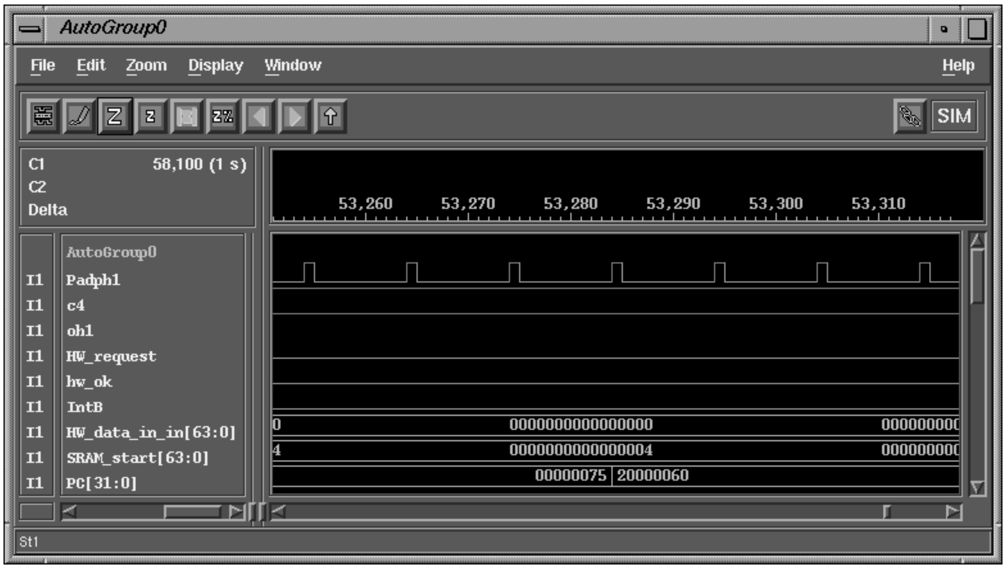


Figure 34: PC Jumps to Start Address for Interrupt Service Routine

In about 30 cycles, as shown in Figure 34, the *PC* jumps to address 0x2000_0060 and start servicing the interrupt. This transfers the contents of *SRAM_start* to the on-chip start vector in register 26.

Completion of a software task

Figure 35 illustrates the sequence of events at the completion of a software task. Here, the software task `cjd` has finished execution sometime prior to time 52,790 and the on-chip done vector in *Reg*27 contains the correct value. The done vector (*done*) in the Run-Time Scheduler hardware still has its old value, however. The *PC* has just finished executing instructions (before 0x9C) that write *Reg*27 to memory-mapped I/O.

Thus *PadAddrValid* goes high as *Reg*27 is first written to the cache. At the same time, the address in memory-mapped I/O where *Reg*27 is going to be written to is on *PadAddress* (ie. 0x2800_4000). Because of the write-through nature of

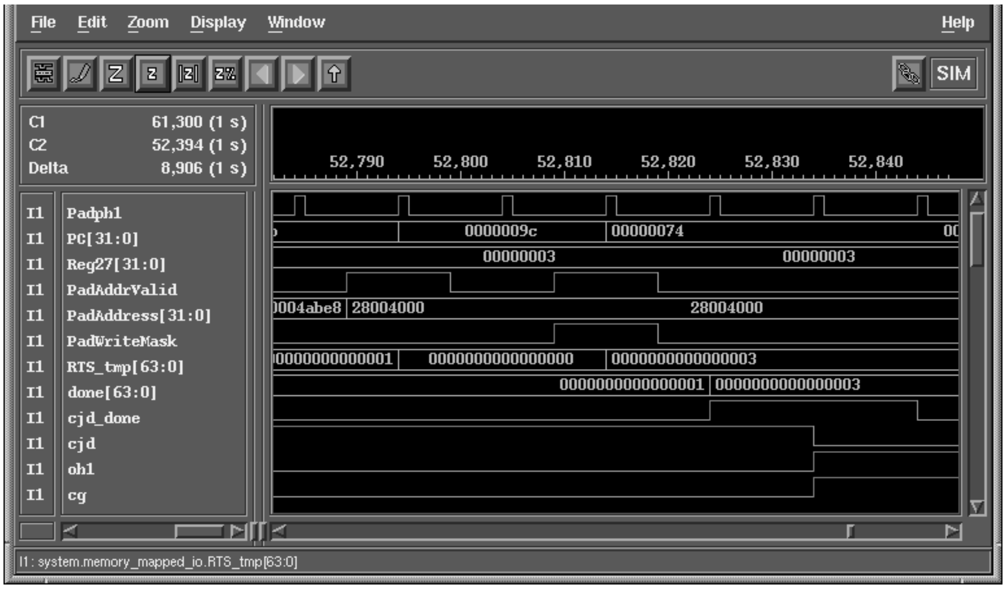


Figure 35: Software task `cjd` completes

the cache, the cache line gets sent to location 0x2800_4000 in memory-mapped I/O; this occurs during the second pulse of *PadAddrValid*. The memory-mapped I/O controller detects the write to 0x2800_4000, recognizes that this is the software done vector and hence writes it out to *RTS_tmp*. A cycle later *done* in *rts.v* gets updated with the correct value. Then *rts.v* asserts *cjd_done*, feeding it to the set ports of the set-reset latches of the next tasks in line (`oh1` and `cg`), causing their *start* vectors to go high. Also, *cjd_done* is routed to the reset port of task `cjd`'s set-reset latch, thus deasserting `cjd`.

Note that the L1 cache and the signals associated with it – *PadAddrValid*, *PadAddress* and *PadWriteMask* – are clocked on *Padph*2 (not shown here).

Reiteration of the control flow

Looking at Figure 36, we see that at the completion of the last task, in this example `xb1`, the control flow loops back and restarts itself. This is predicated by `c0` going high.

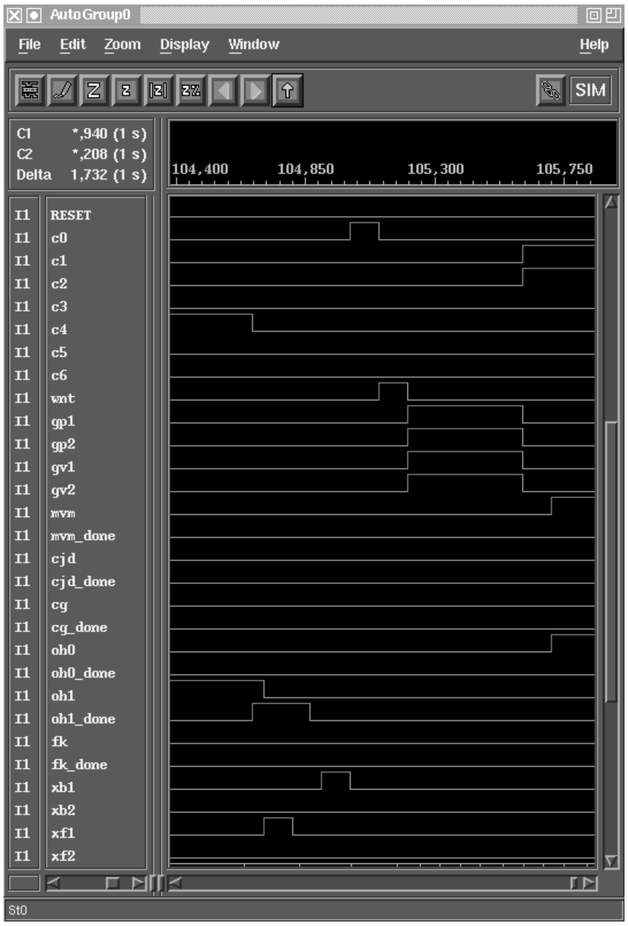


Figure 36: Control restarts itself after task `xb1`

The Run-Time Scheduler smoothly switches between hardware and software tasks. The memory-mapped I/O interface and the interrupt scheme employed to transfer control between hardware and software behave as we predicted. Hardware tasks start and finish at well-defined edges, whereas software tasks are more variable – for instance, there is a distinct lag between the time an interrupt is asserted and the time when program execution starts at the appropriate software task. The Verilog simulation verified the Serra-synthesized run-time scheduler, split between hardware and software, for a controlling two PUMA robot arms.

5.2.6 Design Gains

The original PUMA arm controller was implemented all in C code. However, due to the millisecond timing constraint (a hard real-time constraint), the amount of computation available limits the precision and scope of the control algorithms. For example, designers in the Computer Science Robotics Laboratory at Stanford indicated that while they would very much like to use larger matrices for the state space representation of the kinematics and dynamics of the PUMA arms, they are limited to three-by-three matrices due to the lack of computational power. When they increase the matrix size to five-by-five, the slow down is significant enough to miss the millisecond deadline occasionally. If they increase the matrix size to eleven-by-eleven, then it hardly ever meets the hard real-time constraint of a millisecond.

While the actual hardware-software implementation of the PUMA control system was not carried out, this case study nonetheless shows that such a design could be carried out effectively and to the level of detail required for design space exploration and timing verification. Furthermore, the detailed simulation shows the practicality of the system designed, e.g. in the coordinated flow of *start* and *done* control signals between hardware, software, and the run-time scheduler. With the automation provided by the SERRA system, hardware/software co-design of the control system is greatly improved over manual specification and design of the run-time system. For example, handling multiple $NEVER$ sets of mutually exclusive tasks is much more efficiently dealt with automatically than by hand with the many possible orderings one would need to consider. This is important as the previous research reviewed in Chapter 2 did not handle the case of multiple $NEVER$ sets of tasks in hardware and in software. Overall, the SERRA system can predictably satisfy relative timing constraints, resource constraints (in the form of $NEVER$ sets), and a rate constraint, producing a synthesized run-time system and thus allowing for efficient design space exploration.

5.3 Design Case Study: Haptic Robot

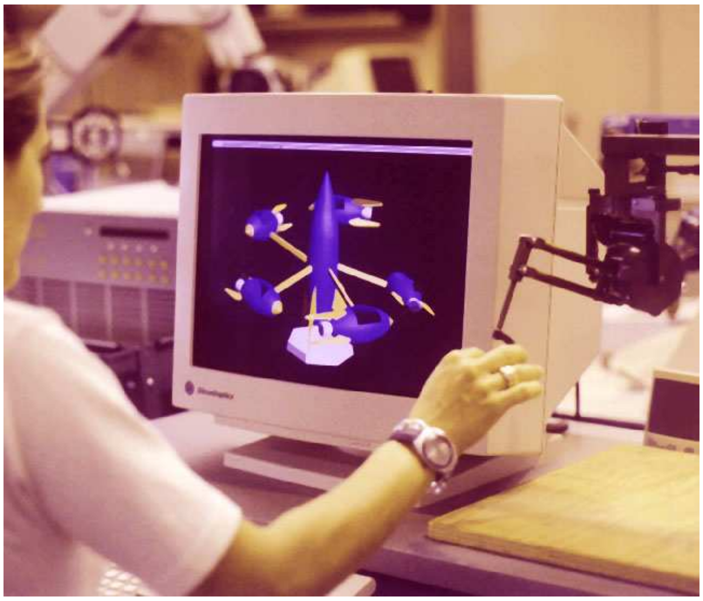


Figure 37: Haptic Robot With Graphics

In this section we present a sample implementation of a split hardware/software run-time scheduler controlling an actual real-time robotics application (as opposed to simulation). We considered the design of the following real-time robotics application: a *Haptic* robot implementing force-feedback based on interaction through a graphics display [RKK97]. The Haptic robotics device contains a thimble where the user places his or her finger. The thimble is connected to the end of a small robot arm which can exert force on the thimble in any direction. The object in the graphics display is represented by a collection of polygons, usually in the range of 10,000 to 20,000 polygons. Figure 37 shows a user interacting with a graphic display where the Haptic device gives feedback based on the position of a small point (called a *proxy*) on the screen. In particular, whenever the proxy collides with a graphical object, a force is generated and the user's finger in the Haptic device is stopped from continuing penetration in that direction. In fact, the feedback is quite complex: the tactile interaction includes contact constraints, surface shading, friction, and texture [RKK97]. Such a system has wide-ranging application possibilities, from helping surgeons operate on patients to training pilots with flight simulation. This application is a good case study because there are some tasks which are poorly implemented in software, e.g. collision detection, which could potentially run much faster in hardware. The first step towards integrating a hardware implementation of such a task into the system is to have a scheduler for the application.

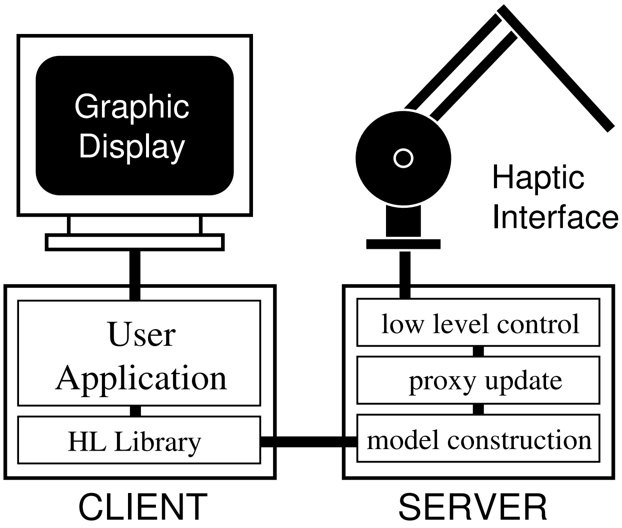


Figure 38: System Architecture

5.3.1 Original Design

The original design consists of an all software solution running on a Silicon Graphics Indigo (SGI) workstation and an IBM compatible PC. The SGI client contains the graphics routines which update the display, and the PC server runs the low level routines for controlling the Haptic device. Our system architecture can be seen in Figure 38.

Collision Detection

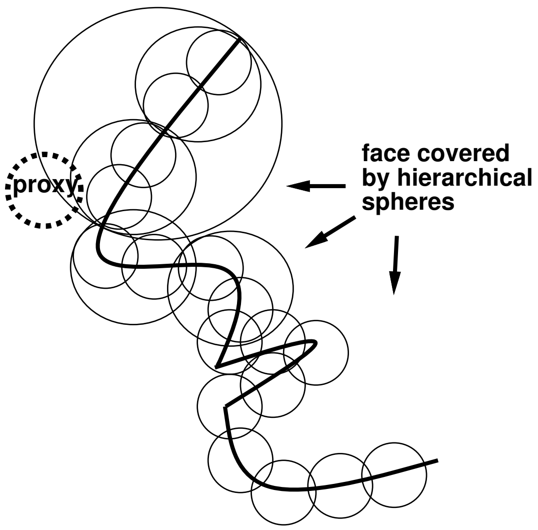


Figure 39: Sphere Characterization

From measurement, we observed that approximately 50% of the CPU time is spent in detecting when the proxy collides with an object in the graphics display. Collision detection is achieved by an algorithmic approach first described in [Qui94]. The basic idea is to take a polygonal surface and cover each polygon with a small sphere. Then, from this initial set of spheres, they are hierarchically covered. Figure 39 shows the beginnings of covering a face using this method (the actual algorithm was written for three dimensions). At the end, we have a root sphere with covers the entire graphical object and all subspheres. The resulting tree data structure of hierarchical spheres has height $O(lg\ n)$. Since the collision detection algorithm checks the sphere hierarchy to see if collision has occured, $O(lg\ n)$ checks are needed.

Timing Constraints

Standard solutions are used for the low level hardware interactions that might otherwise involve strict timing constraints. For writing torque values to the Haptic device, we use a device driver; for reading in the joint positions, we utilize the same device driver to read values from the port.

Model information about the graphics objects and the proxy are communicated between the SGI workstation and the PC by sending and receiving packets using the TCP/IP protocol. In the actual code on the PC, we never perform a blocking wait: instead, we check to see if a packet has arrived, and if so we accept the packet and continue.

The overriding timing constraint we have is a rate constraint: the tasks of the following section must complete before a hard real-time deadline is reached. Any delay in updating the torques could damage the Haptic device or the user.

Haptic Library

The original code for controlling the Haptic device was written in C. Some of the most time-consuming tasks, such as that of communicating the polygons composing the graphics objects and then building a sphere hierarchy, are performed during the initialization and sphere building phases. Once a particular graphical display is up and running, the following tasks are executed in each iteration of a core loop called the *servo* loop:

- wait for next millisecond clock tick
- write torques to Haptic device
- read joint angles of Haptic device
- convert joint angles to x,y,z coordinates
- collision detect
- calculate new proxy position based on collision or not

- compute new torques for Haptic device
- if ready, send/receive network packets (new proxy position, etc.)

For example, consider a user interacting with a graphical display of a teapot. When the proxy is in space not near the teapot, the user can move the proxy freely. However, as soon as the proxy comes close to the teapot, penetrating the sphere hierarchy (an example penetration in two dimensions is shown in Figure 39), collision detection is used to check if the user's proxy on the screen has hit the teapot. The Haptic device provides force-feedback control to simulate the interaction of the proxy with the graphical object, e.g. when sliding along the curved surface of the teapot. Figure 37 shows a user utilizing the proxy to push around a spaceship merry-go-round. An execution of the *servo* loop for controlling the robot must complete once every millisecond.

5.3.2 Haptic Control Implememted with Split Run-Time System

The new design contains a slightly altered scheduler for the *servo* loop. We divide the loop into tasks in order to control their execution from a hardware FSM. Before entering the loop, we kick off execution of the FSM. Within the loop, we execute tasks as directed by the FSM.

Task Execution

We divided the tasks of Section 5.3.1 into three coarse grained groupings as follows:

- "Phantom" routines:
  - wait for next millisecond clock tick

  - write torques to Haptic device
  - read joint angles of Haptic device
  - convert joint angles to x,y,z coordinates

- "Proxy" routines:
  - collision detect
  - calculate new proxy position based on collision or not
  - compute new torques for Haptic device

- "Network" routines, executed only if there are network packets ready to send/receive:
  - send new proxy position to graphics over network
  - receive new graphics info over network

We implemented an FSM in hardware to sequence the above three course granularity software threads. For the sake of experimentation, we use an FPGA-based board (the PCI Pamette[Sha98]) for the hardware implementation. This hardware

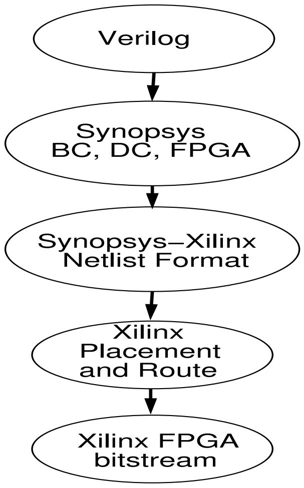


Figure 40: Synopsys-Xilinx Tool Flow

FSM portion of the run-time scheduler is specified in Verilog HDL and synthesized using the Synopsys-Xilinx interface; the tool flow is shown in Figure 40. The Synopsys tools used are the Behavioral Compiler^{TM} (BC)[Kna96], Design Compiler^{TM} (DC) and FGPA Compiler^{TM} (FPGA).

Task execution is described in Section 3.2.1. Briefly, we associate a *start* and a *done* event with each software task (thread). In software, we have a *start* vector and a *done* vector which encapsulate the *start* and *done* events for each software-task. Since there are less than 32 distinct software-tasks, each vector is contained in a single word with a simple one-hot encoding.

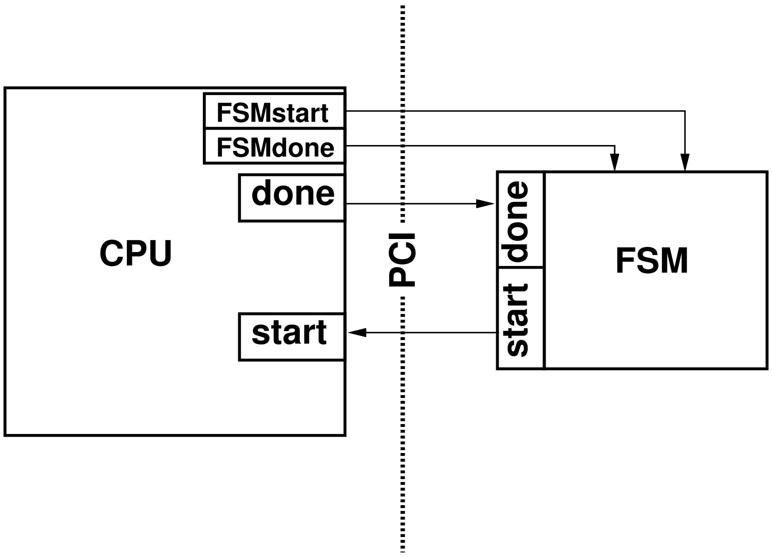


Figure 41: Run-Time Scheduler Control Communication

The run-time scheduler hardware FSM, synthesized to implement the control-flow of task invocations, updates the *start* vector in software as follows. First, it updates a local register containing the *start* vector. Then the CPU reads in the new value on a polling loop. When a software-task is finished executing, it updates the *done* vector by writing the value out with memory mapped I/O. Thus, the the *done* vector in the run-time scheduler in hardware is updated.

Note that we wanted to be able to turn the hardware FSM on and off from software, since the system initialization is directed by software. Thus, we added $FSMstart$ and $FSMdone$ signals to kick off and terminate, respectively, FSM execution. Figure 41 shows the communication of the $FSMstart$, $FSMdone$, $start$ and $done$ vectors.

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 polling scheduler that executes different threads based on eligible software-tasks as indicated by the $start$ vector.

The Haptic library code was altered to accommodate this new split. In particular, a polling scheduler was written as the inner core loop implementing the three course-grained tasks as described here.

The original system in the Computer Science Robotics Lab at Stanford was successfully ported to the NT environment all in software. Then we successfully implemented the split run-time scheduler in the actual design.

5.3.3 System Architecture

Our system architecture consists of an SGI workstation for the graphics, a PC with a Pentium^{TM} processor, and a Haptic device connected to the PC.

The PC has a PCI Pamette[Sha98] board connected to one of its slots. The PCI Pamette, shown in Figure 42, has one FPGA dedicated to talking to the PC using the 32 or 64 bit PCI protocol, with four more Xilinx 4020E FPGAs configurable by the user. The two 128KB SRAMs are essentially scratchpad memories which the nearest FPGA can use. Sixteen bits of memory can be written to or read from each SRAM every cycle.

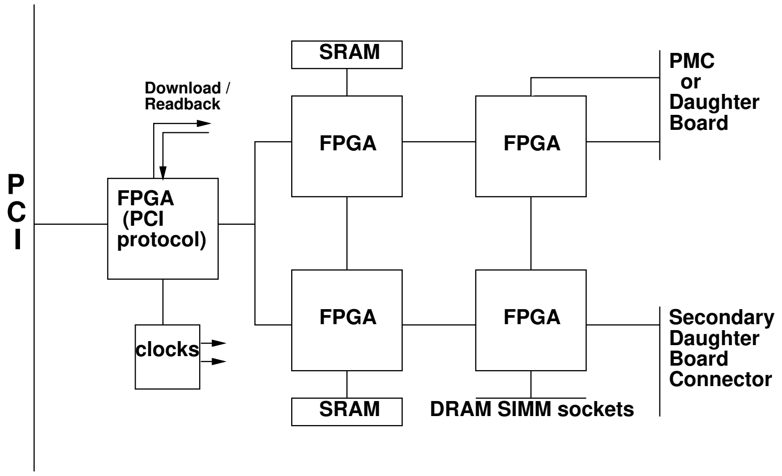


Figure 42: PCI Pamette Version 1 – Architecture

For communication with the FPGAs, we use the PCI protocol as implemented by the PCI Pamette software library for Visual C++ and the FPGA on the PCI Pamette. From the point of view of the software code, this appears as a memory-mapped read or write. However, there are timing constraints which must be observed by the two FPGAs that can read data from the 32-bit bus coming out from the FPGA implementing the PCI protocol: once an address appears on the bus, the data corresponding to that address must be read in the following cycle. Similarly, for writing to the bus (in which case the software is executing a read from memory-mapped I/O), the data read must be driven to the bus on the following cycle and held there for six cycles. There are many more constraints explained in the PCI documentation [Sha98].

In order to meet these exact timing constraints, we latch values going on/off chip using DC^{TM} and then read the values using behavioral Verilog synthesized in cycle-accurate mode with BC^{TM}.

5.3.4 Software Generation

The software for programming and controlling the PCI Pamette is available for Microsoft Visual C++ 4.0^{TM} with Windows NT 4.0^{TM} or for the DEC Alpha. Because we wanted to use a PC, we utilized the NT version.

The original code (called the "Haptic library") for controlling the Haptic device was written in 10,000 lines of C for Linux. In order to use the Pamette, we ported the Haptic library to Visual C++ 4.0^{TM} with Windows NT 4.0^{TM}. This porting effort included writing a device driver in NT to control the Haptic device as well as rewriting the network code for communication with the SGI workstation using TCP/IP.

Reading and writing to the SRAM on the Pamette is accomplished using memory-mapped I/O and hardware-tasks in the FPGA. The PCI interface takes an average of 5 to 9 CPU clock cycles to communicate a single 32-bit read or write.

Therefore, given a particular value of the *start* vector, the appropriate software-task(s) can be executed. The scheduler for the software is a simple polling loop. Note that for this to work we have to guarantee that after indicating that a particular

Figure 43: Teapot Graphical Object With Proxy

software-task has completed by writing to the *done* vector, the next *start* value must be updated and ready to be read before the software polling loop next reads in the *start* vector. Otherwise, the software scheduler could read in the exact same *start* vector again and thus fail to meet the rate constraint of updating the robot's torque values every millisecond. We verified that the FSM implemented in the FPGA was fast enough by extensive simulation.

Figure 43 shows a graphical teapot model which we used to test the design. The proxy is shown on the teapot near the base of the spout. The teapot is composed of 3,416 triangular surfaces. The client computer was an SGI Indigo2 High Impact running IRIX 6.2 and the Haptic server was a PC with a 266 Mhz Pentium Pro running Windows NT 4.0. The PC has 32 MB of main memory and a 512KB cache. Communication between the two computers was done through a standard ethernet TCP/IP connection. The Haptic device used was a ground based PHANToM manipulator with 3 degrees of freedom in it force-feedback.

Task	Lines C
wait for next millisecond clock tick	65
write torques to Haptic device	50
read joint angles of Haptic device	48
convert joint angles to x,y,z coordinates	428
collision detect	2189
calculate new proxy position	664
compute new torques for Haptic device	10
send/receive information over network	1328
device driver	899

Table 11: Code space for software tasks.

Table 11 shows the code space used for the various software tasks in the inner *servo* loop. The final executable took up 485KB of memory; however, the code and

Task	Lines Verilog	Style of Verilog
ebusread1.v	146	behavioral
ebuswrite1.v	114	behavioral
generatecontrol.v	48	behavioral
haptic.v	242	structural
hapticcontrol.v	178	behavioral
startcontrol.v	150	behavioral
transactionmodelib.v	154	structural
writestart.v	99	behavioral

Table 12: Code space for hardware tasks.

data used in the *servo* loop is much less and likely fit entirely in the 512KB cache on the PC (however, we did not verify this). Table 12 shows the code space used for reading and writing data from/to the bus and the SRAM, starting/terminating the hardware FSM, and the hardware FSM itself (in `hapticontrol.v`). Notice that the FSM takes only 178 lines of Verilog HDL, while the supporting Verilog HDL code takes 1195 lines. Table 13 shows the various measures of utilization provided for the Xilinx 4020E which implements the Verilog HDL code. The 4020E can fit at most around 20K logic gates. We are currently using about half of the available CLBs.

5.3.5 Future Directions

For future work, an ASIC implementation of the collision detection algorithm would drastically speed up the application, especially since the sphere checking is quite naturally parallelizable. The run-time scheduler described here could quite easily be augmented with such an ASIC. In fact, the inclusion of multiple components in hardware could be easily added to the system. The major practical design cost would be the specification and design of the collision detection ASIC.

The PC-Pamette architecture described in the previous sections provides the basis

Xilinx Measure	No. Used	Max. Avail.	Percent Used
Occupied CLBs	401	784	51%
Bonded I/O Pins	72	160	45%
F and G Function Generators	494	1568	31%
H Function Generators	93	784	11%
CLB Flip Flops	217	1568	13%
IOB Input Flip Flops	33	224	14%
IOB Output Flip Flops	18	224	8%
3-State Buffers	0	1680	0%
3-State Half Longlines	0	112	0%
Edge Decode Inputs	0	336	0%
Edge Decode Half Longlines	0	32	0%
CLB Fast Carry Logic	8	784	1%

Table 13: Statistics for Xilinx 4020E Mapping

for a modular extensible hardware-software run-time system. Since the hardware part of the run-time system is in FPGAs, it can be reconfigured quickly with the synthesis path of Figure 40. Currently we only use one of the four available FPGAs. Portions of the real-time Haptic control system can be migrated to hardware, either into FPGAs, ASICs or DSPs. For example, an ASIC implementing the collision detection algorithm (which has a lot of parallelism) could be integrated quickly into the run-time system.

For the final embedded application, the hardware part of the run-time system is synthesized into hardware rapidly since it is described in behavioral Verilog and uses synthesis all the way down to the bitstream for programming the XILINX 4020E FPGAs. For example, given a working protype, one could design a single chip implementation of the control system using a Pentium core, dedicated logic for the logic implemented in FPGAs in the prototype, and a core for the ASIC implementing the collision detection algorithm. In other words, given the Intellectual Property (IP) for each component used in the prototype, it is possible that the entire design could be placed on the same single chip and fabricated.

5.4 Summary

In conclusion, we have shown two sample applications of a run-time scheduler split between hardware and software. The first verified via simulation the results of using the SERRA system on a robot arm controller for two PUMA arms. The second successfully implemented the split run-time scheduler approach on a real-time Haptic robot prototype.

Note the CAD requirements for hardware/software co-design of both systems. First of all, we needed to satisfy a hard real-time relative scheduling constraint imposed by two of the hardware components in the PUMA control system: tasks `xf1` and `xb1` had a relative timing constraint of activating the *start* event for `xb1` no less than 2 and no greater than 8 cycles after activating the *start* event for `xf1`. This constraint was always satisfied in any of the run-time systems generated by SERRA. Satisfaction of this constraint was a *sine qua non* for exploring different hardware/software partitions of the PUMA system, which occurred as we moved some tasks from hardware to software and vice-versa. Second, we altered the $NEVER$ sets as we moved tasks around between hardware and software. SERRA's run time system, when it found a solution, always satisfied the mutual exclusion required by the specified $NEVER$ sets. Notice that the *WCET* values for the different tasks can be quite large; thus, the effect of high-level decisions as to the hardware/software partition as well as the amount of hardware dedicated to each hardware task (the more hardware area allowed would typically result in a faster hardware task) could be evaluated quickly and in an interactive fashion. Thus, the designer was able to optimize that design by exploring the design space much more effectively than if the run-time system had to be regenerated by hand for each partition. Furthermore, the automatic satisfaction of specified timing constraints and resource constraints freed the designer from time consuming calculations that can be automated using the approach in this thesis.

The real-time Haptic robot prototype showed the feasibility of the run-time scheduler approach of this thesis in an actual working design. Where as the PUMA controller example explored the range of options afforded by the SERRA system, such as the support for timing and resource constraints, the Haptic robot prototype focused on the practicality of the design style. Thus, the split run-time scheduler functioned perfectly well in the final prototype.

In short, both the PUMA controller simulation and the Haptic robot prototype show the feasibility and utility of the design style and CAD tool for extensible run-time systems in hardware and software proposed in this thesis.

Chapter 6

Conclusions and Future Work

6.1 Summary

We considered in this thesis analysis and synthesis techniques for hardware/software run-time systems. We assume a system specification at the task level of hardware and software, e.g. hardware modules and software threads, together with a main task that specifies the control and data flow among all of the specified tasks. We further assume a target architecture of a CPU core to run the software-tasks together with custom hardware implementing the hardware-tasks. Unlike previous approaches to run-time scheduling, we split our run-time scheduler between hardware and software, as opposed to placing the scheduler all in one or the other. Our analysis also takes into account the split hardware/software implementation both of the scheduler and of the tasks.

Thus, we have presented a new system-level scheduling methodology and CAD tool for hardware/software co-design. In particular, we focused on the following aspects of hardware/software run-time system analysis and synthesis:

Design Style for Scheduling. We presented a design style for synchronization and scheduling in hardware/software co-design where we represent each hardware- and software-task by an automaton that begins execution upon receiving a *start* event and indicates that it has completed execution by emitting a *done* event. These *start* and *done* events are appropriately implemented in hardware and software.

Co-Synthesis of a Hardware-Software Run-Time Scheduler. We showed how, given the control-flow of the tasks in the system, a run-time scheduler can be synthesized in hardware and software. The hardware part of the run-time scheduler consists of an FSM in hardware sequencing the *start* and *done* events such that all specified relative timing constraints are met; relative timing constraint satisfaction is assured by construction in that we only generate an FSM if we will meet the relative timing constraints. The software part of the run-time scheduler is a small kernel consisting of an interrupt service routine, context switch code, and a static preemptive priority scheduler. The priority scheduler is parameterized to fit the exact number of software-tasks in the system. This mixed implementation leverages advantages of hardware, such as predictable and exact fine-grained timing separation of *start* signals, and advantages of software, such as preemptibility and flexibility.

Rate-Constraint Satisfaction Analysis. We presented several scheduling algorithms for tasks in hardware and software under a hard real-time rate constraint where the precedence constraints among tasks are specified in a DAG. In effect, these techniques extend *Worst-Case Execution Time* analysis from the pure software and pure hardware domains to a mixed hardware/software implementation domain. Furthermore, with our small, custom kernel for the software operating system, we can analyze rate-constraint satisfaction in cases where task preemption may occur.

Resource Constraints. The scheduling style and algorithms support a $NEVER$ set of software-tasks implemented on the same CPU and multiple $NEVER$ sets of hardware-tasks. While the run-time scheduler can trivially maintain mutual exclusion among tasks in the same $NEVER$ set by the addition of precedence constraints, rate-constraint analysis gives a set of precedence constraints to add to the scheduler which yields the smallest *WCET* for the system among the task orders considered.

Application to Robotics. We presented a full analysis, synthesis, and simulation of a hardware/software implementation of a system for controlling two PUMA robot arms. We also described a small prototype of a split hardware/software run-time scheduler to control a force-feedback Haptic robot using a Pentium^{TM} for the software and Xilinx FPGAs for the hardware.

The approach of this thesis allows the exploration of the design space of a run-time scheduler across the boundary of hardware and software. The designer gains the advantage of predictable satisfaction of timing constraints for hardware/software systems within a framework that enables different hardware/software partitions to be quickly evaluated. Thus, in relation to previous work in hardware/software partitioning, a partitioning tool could easily sit on top of SERRA which would generate run-time systems for different hardware/software partitions chosen for evaluation. In addition, SERRA's more efficient design space exploration can improve time-to-market for a product, enabling the product to enter the market sooner.

6.2 Future Work

During the development of this work, we observed that several lines of research which promise significant impact in the coming years.

- Hardware/software co-design of a modular Real Time Operating System in hardware and software. With so much space available on the chip, future system-on-a-chip designs will provide an opportunity to redefine the goals of RTOS research for embedded systems. In particular, with the high demands placed on tomorrow's dedicated applications, there is a role for an RTOS where hardware is used to both speed up critical bottlenecks and more effectively manage concurrency among hardware and software. Such a modular RTOS would include providing functionality for updating and debugging the system, functionality which is "modular" and thus can be thrown away when the real application is to be run at the highest possible performance.

- Reconfigurable architectures for application-specific system-on-a-chip designs. For a set of applications, a system-on-a-chip design can be tuned to maximize performance for the specific applications desired. Reconfigurable interfaces among tasks as well as reconfigurable scheduling can help obtain the best results. A design style with associated CAD tools for such application-specific reconfigurable architectures can key significant improvements in design space exploration, time-to-market, and even testability of the final chip.

- Hardware/software co-design of reconfigurable run-time systems with multiple processors. If the FSM part of the run-time scheduler presented in this thesis were implemented in reconfigurable logic, for example with SRAM-based FPGA technology, then the hardware part of the scheduler would be reconfigurable.

The software part is already reconfigurable by definition. With multiple processors available, e.g. a DSP core and a microcontroller core, the design space to be explored is quite significant. Greatly needed are CAD tools to allow designers to efficiently explore this wide design space.

Abbreviations and Symbols

□ Symbol marking the end of an example.

♯ Symbol marking the end of a proof.

AMPL A Mathematical Programming Language

ASIC Application Specific Integrated Circuit

BC^{TM} Synopsys Behavioral Compiler^{TM}

BCET Best-Case Execution Time

BDD Binary Design Diagram

CAD Computer-Aided Design

CB Communication Blocks

CFSM Co-design Finite State Machine

CDFG Control-Data Flow Graph

CPU Central Processing Unit

DAG Directed Acyclic Graph

DFL Data Flow Language

DC^{TM} Synopsys Design Compiler^{TM}

DMA Direct Memory Access

DSP Digital Signal Processing

FCFS First-Come-First-Served

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

HDL Hardware Description Language

IBC Inter-Block Communication

IC Integrated Circuit

ILP Integer Linear Program

I/O Input/Output

IP Intellectual Property

LCM Least Common Multiple

NSDS Never Set DAG Scheduling

PCI Peripheral Component Interconnect bus for ASIC designers

RAM Random Access Memory

RMA Rate Monotonic Analysis

RPC Remote Procedure Call

RTL Register-Transfer-Level

RTOS Real-Time Operating System

SoC System-on-a-Chip

SRAM Static Random Access Memory

SRTD Sequencing with Release Times and Deadlines

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

WCET Worst-Case Execution Time

Bibliography

[ABD$^+$95] 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.

[ABR$^+$93] 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*, pages 284–292, 1993.

[ACJ96] M. Abid, A. Changuel, and A. Jerraya. "Exploration of Hardware/Software Design Space through a Codesign of Robot Arm Controller". In *Proceedings of the European Design Automation Conference*, pages 42–47, September 1996.

[AFLS96] J. Adomat, J. Furunas, L. Lindh, and J. Starner. "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High Performance Real-Time Systems". In *Real-Time Workshop*, June 1996.

[AKB86] B. Armstrong, O. Khatib, and J. Burdick. "The explicit model and inertial parameters of the PUMA 560 arm". *Proceedings of IEEE International Conference on Robotics and Animation*, 1:510–518, 1986.

[AM98] A. Morawiec, President. European CAD Standardization Initiative. *http://www.vsi.org/*, 1998.

[AT95] J. Adams and D. Thomas. "Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems". In *Proceedings of the International Symposium on System Synthesis*, pages 10–15, September 1995.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[BCO96] G. Borriello, P. Chou, and Ross B. Ortega. "Embedded System Co-Design: Towards Portability and Rapid Integration". In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 243–264. Kluwer Academic Publishers, 1996.

[Ber96] G. Berry. See http://zenon.inria.fr/meije/esterel/, 1996.

[BG] G. Berry and G. Gonthier. *"The Esterel Synchronous Programming Language: Design, Semantics, Implementation"*. Ecole Nationale Supérieure des Mines de Paris and Institut National de Recherche en Informatique et Automatique.

[BHLM94] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, April 1994. *Also available from http://ptolemy.eecs.berkeley.edu*.

[BS91] F. Boussinot and R. De Simone. "The ESTEREL Language". *Proceedings of the IEEE*, 79(9):1293–1303, September 1991.

[CB94] P. Chou and G. Borriello. “Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems”. In *Proceedings of the* 31^{st} *Design Automation Conference*, pages 1–4, June 1994.

[CM96] C. N. Coelho Jr. and G. De Micheli. “Analysis and Synthesis of Concurrent Digital Circuits Using Control-Flow Expressions”. *IEEE Transactions on CAD/ICAS*, 15(8):854–876, August 1996.

[CM97] C. N. Coelho Jr. and G. De Micheli. “Modeling and Synthesis of Synchronous System-Level Specifications”. In J. Berge, O. Levia, and J. Rouillard, editors, *Models in System Design*, pages 243–264. Kluwer Academic Publishers, 1997.

[COB92] Pai Chou, Ross Ortega, and Gaetano Borriello. “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”. In *Proceedings of the International Conference on Computer-Aided Design*, pages 488–495, Santa Clara, November 1992.

[COB95] P. Chou, Ross B. Ortega, and G. Borriello. “The Chinook Hardware/Software Co-Synthesis System”. In *Proceedings of the International Symposium on System Synthesis*, pages 22–27, September 1995.

[Coe96] C. N. Coelho Jr. “Analysis and Synthesis of Concurrent Digital Systems Using Control-Flow Expressions”, March 1996. CSL-TR-96-690.

[Cow98] CoWare touts ‘interface synthesis’ for codesign. *EE Times*, page 54, February 1998.

[CWB94] P. Chou, E. Walkup, and G. Borriello. “Scheduling for Reactive Real-Time Systems”. *IEEE Micro*, August 1994.

[DF98] D. Fairbairn, President. Virtual Sockets Interface Alliance. *http://www.vsi.org/*, 1998.

[DJ98] B. Dave and N. Jha. "CASPER: Concurrent Hardware-Software Co-synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Architectures". In *Proceedings of the Design, Automation and Test in Europe*, pages 118–124, February 1998.

[DKMT90] G. DeMicheli, D. C. Ku, F. Mailhot, and T. Truong. "The Olympus Synthesis System for Digital Design". *IEEE Design and Test*, pages 37–53, October 1990.

[DLJ97] B. Dave, G. Lakshminarayana, and N. Jha. "COSYN: Hardware-Software Co-synthesis of Embedded Systems". In *Proceedings of the Design Automation Conference*, pages 703–708, June 1997.

[EHB+96] 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". *IEEE Micro*, 20:159–166, 1996.

[EKP+98] P. Eles, K. Kucheinski, Z. Peng, A. Doboli, and P. Pop. "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems". In *Proceedings of the Design, Automation and Test in Europe*, pages 132–138, February 1998.

[EY97] R. Ernst and W. Ye. "Embedded program timing analysis based on path clustering and architecture classification". In *Proceedings of the International Conference on Computer-Aided Design*, pages 598–604, Santa Clara, CA, November 1997.

[FGK93] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, 1993.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability*. W. Freeman and Company, 1979.

[Gup95] R. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, 1995.

[HB97] Ken Hines and Gaetano Borriello. "Optimizing Communication in Embedded System Co-simulation". In *International Workshop on Hardware/Software Co-Design*, pages 121–125, 1997.

[HE95] J. Henkel and R. Ernst. "A Path-Based Technique for Estimating Hardware Runtime in HW/SW-Cosynthesis". In *Proceedings of the International Symposium on System Synthesis*, pages 116–121, September 1995.

[HE96] J. Henkel and R. Ernst. "The Interplay of Run-Time Estimation and Granularity in HW/SW Partitioning". In *International Workshop on Hardware/Software Co-Design*, 1996.

[HL95] F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 1995.

[HWS95] M. Humphrey, G. Wallace, and J. Stankovic. "Kernel-Level Threads for Dynamic, Hard Real-Time Environment". In *Proceedings of the Real Time Systems Symposium*, pages 38–48, 1995.

[KD90] D. C. Ku and G. DeMicheli. HardwareC - a language for hardware design (version 2.0). CSL Technical Report CSL-TR-90-419, Stanford, April 1990.

[KM92] D. Ku and G. De Micheli. *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[Kna96] D. Knapp. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Prentice-Hall, 1996.

[Lat91] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Lin92] L. Lindh. "Idea of FASTHARD - A Fast Time Deterministic Hardware Based Real-Time Kernel". In *Real-Time Workshop*, June 1992.

[LL73] C. Liu and J. Layland. "Scheduling algorithms for multiprogramming in a hard-real time environment". *Journal of the ACM*, 20(1):46–61, 1973.

[LM95] Y. Li and S. Malik. "Performance Estimation of Embedded Software with Instruction Cache Modeling". In *Proceedings of the International Conference on Computer-Aided Design*, pages 380–387, Santa Clara, CA, November 1995.

[LS91] L. Lindh and F. Stanischewski. "FASTCHART – Idea and Implementation". In *Proceedings of the International Conference on Computer Design*, pages 401–404, 1991.

[LSF95] L. Lindh, J. Starner, and J. Furunas. "From Single to Multiprocessor Real-Time Kernels in Hardware". In *Real-Time Technology and Applications Symposium*, May 1995.

[MBL+96] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. "Co-design of DSP Systems". In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 75–104. Kluwer Academic Publishers, 1996.

[MBL+97] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. "Hardware/Software Co-Design of Digital Telecommunication Systems". *Proceedings of the IEEE*, 85(3):391–418, March 1997.

[MCSM96] V. J. Mooney III, C. N. Coelho Jr., T. Sakamoto, and G. De Micheli. Synthesis from mixed specifications. In *Proceedings of the European Design Automation Conference*, pages 114–119, September 1996.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[Mic97] G. De Micheli. Special Issue on Hardware/Software Co-Design. In *Proceedings of the IEEE*, March 1997.

[MS96] G. De Micheli and M. Sami. *Hardware/Software Co-Design.* Kluwer Academic Publishers, 1996.

[MSM97] V. J. Mooney III, T. Sakamoto, and G. De Micheli. "Run-Time Scheduler Synthesis For Hardware-Software Systems and Application to Robot Control Design". In *International Workshop on Hardware/Software Co-Design*, pages 95–99, March 1997.

[MWWL96] S. Malik, W. Wolf, A. Wolf, and Y. Li. "Performance Analysis of Embedded Systems". In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 45–74. Kluwer Academic Publishers, 1996.

[NVG92] S. Narayan, F. Vahid, and D. Gaski. "System Specification with the SpecCharts Language". *IEEE Design & Test of Computers*, pages 6–13, December 1992.

[OB97] Ross Ortega and Gaetano Borriello. "Communication Synthesis for Embedded Systems with Global Considerations". In *International Workshop on Hardware/Software Co-Design*, pages 69–73, 1997.

[OBE+97] A. Osterling, T. Benner, R. Ernst, D. Herrmann, T. Scholz, and W. Ye. "The Cosyma System". In *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[PS89] D. Peng and K. Shin. "Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-Time Systems". In *International Conference on Distributed Computing Systems*, pages 190–198, 1989.

[Qui94] S. Quinlan. "Efficient Distance Computation between Non-Convex Objects". *International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[Raj91] R. Rajkumar. *"Synchronization in Real-Time Systems: A Priority Inheritance Approach"*. Kluwer Academic Publishers, 1991.

[Ram90] K. Ramamritham. "Allocation and Scheduling of Complex Periodic Tasks". In *International Conference on Distributed Computing Systems*, pages 108–115, 1990.

[Ram95] K. Ramamritham. "Allocation and Scheduling of Precedence-Related Periodic Tasks". *IEEE Proceedings on Parallel and Distributed Systems*, 6(4):412–420, April 1995.

[RK98] Ryo Koyama, Chairman, Board of Directors. Reuseable Application-Specific Intellectual Property Developers. *http://www.rapid.org/*, 1998.

[RKK97] D. Ruspini, K. Kolarov, and O. Khatib. "The Haptic Display of Complex Graphical Environments". *Proceedings of SIGGRAPH*, pages 345–352, August 1997.

[RVBM96] Karl Van Rompaey, Diederik Verkest, Ivo Bolsens, and Hugo De Man. "CoWare – A design environment for heterogeneous hardware/software systems". In *Proceedings of the European Design Automation Conference*, pages 252–257, September 1996.

[Sha98] M. Shand. PCI Pamette V1. *Digital Equipment Corporation, System Research Center, http://www.research.digital.com/SRC/pamette/*, 1998.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronizations". *IEEE Transactions on Computers*, pages 1175–1185, December 1990.

[SRS94] 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.

[SSM+92] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuits design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, Cambridge, MA, 1992.

[SSNB95] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, pages 16–47, June 1995.

[Uni84] Unimation. "Unimate PUMA Mark II Robot: 500 Series Equipment and Programming Manual 398P1". pages 1–36, April 1984.

[VLM96a] S. Vercauteren, B. Lin, and H. De Man. "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications". In *Proceedings of the Design Automation Conference*, pages 521–526, June 1996.

[VLM96b] S. Vercauteren, B. Lin, and H. De Man. "Embedded Architecture Co-Synthesis and System Integration". In *International Workshop on Hardware/Software Co-Design*, 1996.

[VRBM96] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man. "CoWare – A Design Environment for Heterogeneous Hardware/Software Systems". *Design Automation of Embedded Systems*, 1(4):357–386, October 1996.

[WDC+94] P. Willekens, D. Devisch, M. Van Canneyt, P. Conflitti, and D. Genin. "Algorithm Specification in DSP Station using Data Flow Language". *DSP Applications*, pages 8–16, January 1994.

[YW95] T.-Y. Yen and W. Wolf. "Performance Estimation for Real-Time Distributed Embedded Systems". In *Proceedings of the International Conference on Computer Design*, pages 64–69, 1995.

[YW96] T.-Y. Yen and W. Wolf. *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. Kluwer Academic Publishers, 1996.

Appendix A

A Mathematical Program Formulation

In this section we show a mathematical program formulation for optimal scheduling of hardware and software tasks in a DAG with a single $NEVER$ set. Specifically, we show the formulation as implemented in the AMPL modeling language [FGK93]. We

```
set DAG := src a b c d e f snk;
set N := b c d;
# set of Predecessors
set P := (src,a) (src,b) (src,c) (a,d) (b,e) (d,f) (c,snk) (e,snk) (f,snk);

param:   WCET :=
src      0
a        5000
b        3000
c        20000
d        15000
e        5000
f        11000
snk      0 ;
```

Figure 44: AMPL data for dagopt problem.

then compare the AMPL solution to our solution using the Constructive Heuristic Scheduling of Section 4.2.

For our example, we use the DAG of Figure 16 and refer to it as the `dagopt` problem. For our mathematical model in AMPL shown in Figure 45, we define the set DAG to contain the nodes shown in Figure 16, including the source and the sink. The set N contains the tasks in the same $NEVER$ set; in this example we have a single $NEVER$ set. The set P contains the set of predecessors: $(x, y) \in P$ indicates

```
set DAG;                               # nodes in Directed Acyclic Graph
set N;                                 # NEVER set
set P within (DAG cross DAG);          # set of predecessors, starting with the src

param WCET DAG >= 0;                   # WCET for each node in DAG
param MAXTIME >= 0 default 1000000;
var x i in N, j in N binary;           # = 1 if task j is processed before i, 0 o.w.
var starttime DAG >= 0;                # time when each node starts
var sinkendtime >= 0;

minimize max_cost: sinkendtime;

subject to sinkendtime_def i in DAG:
    sinkendtime >= starttime[i] + WCET[i];

subject to Precedence_s (i,j) in P:
    starttime[j] >= starttime[i] + WCET[i];

subject to Mutual_exclusion i in N, j in N: i != j:
    starttime[j] + WCET[j] <= starttime[i] + (1 - x[j,i])*MAXTIME;

subject to Mutual_exclusion2 i in N, j in N: i != j:
    starttime[j] + x[j,i]*MAXTIME >= starttime[i] + WCET[i];

subject to same_x2 i in N:
    x[i,i] = 1;
```

Figure 45: AMPL model for `dagopt` problem.

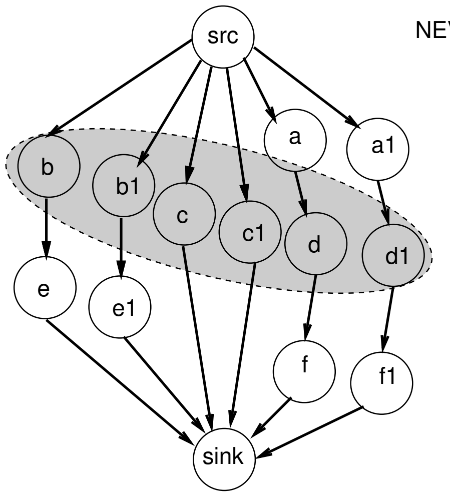
that x is a predecessor of y in the DAG. With each element of the set DAG, we associate a *WCET* in the set WCET, which is defined over the elements of DAG. The binary decision variable x decides the order of tasks in the $NEVER$ set, set N. Constraint PRECEDENCE_S makes sure that no task starts until all of its predecessors have completed execution. Constraints MUTUAL_EXCLUSION and MUTUAL_EXCLUSION2 make sure that for two tasks i and j in set N, $i \neq j$, **either** task i finishes execution before task j, **or** task j finishes execution before task i. The MAXTIME constant indicates that above MAXTIME cycles, we do not need to look anymore, because our real-time constraint is violated. The use of the MAXTIME constant enables the formulation of linear constraints in the AMPL model. The data corresponding to this model is shown in Figure 44.

We solve the problem in AMPL using the solver CPLEX [FGK93]. CPLEX uses an exact solution method to find the optimal schedule, which results in a *WCET* of 40,000 cycles. The solution found with Constructive Heuristic Scheduling, however, finds a suboptimal schedule resulting in a *WCET* of 43,000 cycles, exactly as described in Figure 16.

Example	WCET w/ heuristic	ET w/ heuristic	WCET w/ AMPL	ET w/ AMPL
dagopt	43,000	0.003 seconds	40,000	0.03 seconds
dagopt2	81,000	0.010 seconds	76,000	1.6 seconds
dagopt3	119,000	0.040 seconds	114,000	24 minutes
dagopt4	157,000	0.110 seconds	—	> one day
dagopt5	195,000	0.280 seconds	—	out of memory

Table 14: WCET found and run times for Constructive Heuristic Scheduling versus AMPL.

This base example, which we label `dagopt` in Table 14, has been extended to four additional examples simply by doubling, tripling, quadrupling, and quintupling the number of tasks in `dagopt` – the examples are named `dagopt2`, `dagopt3`, `dagopt4` and `dagopt5`, respectively. For example, `dagopt2` is shown in Figure 46. We find, as expected, that the polynomial algorithm of the Constructive Heuristic Scheduling performs faster than the exact algorithm of CPLEX. Furthermore, in `dagopt4` and `dagopt5`, which have 26 and 32 nodes, respectively, we find that CPLEX does not even arrive at a solution after one day. The resulting *WCET*'s, each with a corresponding `ET` (Execution Time), are shown in Table 14. The Constructive Heuristic Scheduling algorithm was run on a Silicon Graphics INDY 4400 at 200 MHz with 64 MBytes of RAM. The AMPL program was executed on a Sun SPARCstation 20 at 150 MHz with 64 MBytes of RAM.



NEVER = {b,c,d,b1,c1,d1}

task	wcet (cycles)
a	5,000
b	3,000
c	20,000
d	15,000
e	5,000
f	11,000
a1	5,000
b1	3,000
c1	20,000
d1	15,000
e1	5,000
f1	11,000

Figure 46: The `dagopt2` problem, generated from the `dagopt` problem (Figure 16) by doubling the number of tasks.

Of course, AMPL/CPLEX can solve an extremely wide range of optimization problems exactly, whereas the Constructive Heuristic Scheduling algorithm is targeted to a specific problem.