Hardware/Software Partitioning and Scheduling of Embedded Systems

by

Andrew Morton

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

Abstract

Hardware/software codesign is a design methodology for embedded systems that seeks to satisfy system-level constraints by exploiting the synergy between hardware and software through their concurrent design [71]. During partitioning, design components are assigned to hardware and software implementation targets. The output of the partitioner has a significant impact on the subsequent scheduling of software. This dissertation is a study of the relationship between the partitioning and scheduling of concurrent systems as it pertains to hardware/software codesign. It is structured as three related studies.

The first study is on the Earliest Deadline First (EDF) scheduling policy. This is an optimal policy that yields high processor utilization. An algorithm for the off-line feasibility analysis of systems scheduled by EDF is extended in this study to accommodate tasks that employ coprocessors during their execution. This is relevant to software systems that have been partitioned by moving part of the application functionality into hardware coprocessors.

The second study is a case study of a real-time kernel that implements the EDF scheduling policy and an automotive application executing on a System on Programmable Chip (SoPC). The application is partitioned by creating a hardware coprocessor. The kernel is also partitioned. The impact on the schedule feasibility of the system is examined for both coprocessors. A metric is also proposed that facilitates comparison of coprocessors, whether for the application or the kernel.

The third study is of automated partitioners. The hardware/software partitioning problem is defined such that it includes schedule feasibility and the joint partitioning of application and kernel. A non-linear programming model is created to capture the problem definition and a heuristic is developed based on the Fiduccia/Mattheyses heuristic. Results confirm that the scheduling problem is hard enough that it is not likely to be satisfied unless directly addressed by the partitioner. The results also show that including the kernel in partitioning helps to produce solutions with feasible schedules.

The inter-related nature of hardware/software partitioning, and scheduling of concurrent systems is demonstrated through the three studies described above. In the process, scheduling by the preemptive Earliest Deadline First policy is explored in its relation to hardware/software codesign. It is shown that scheduling of concurrent systems needs to be integrated into hardware/software partitioning in order to meet demanding system constraints. Since all three studies focused on single-processor systems, an avenue for further exploration is to consider multi-processor systems.

Acknowledgments

I will strength you and help you; I will uphold you with my righteous right hand. — Isaiah 41:10

My first thanks are to my wife Rachel - you were very encouraging. I'd also like to thank Abigail and Stephen for many enjoyable distractions. I'm also thankful for our third child who is currently on the way. Thanks to all of my family - Mom, Dad, Heather, Jen, Jon, Aunt Joyce, Uncle Tom, Grandma and Grandpa. Thanks to Rachel's family and especially Mom R for her prayers.

Thank-you Wayne - I am glad that I got to work with you. You were patient with me.

Thank-you Tony Vannelli for the financial support which was so important.

Thank-you Gary Grewal, my wise friend in academia who kept bugging me to publish!

Thanks Paul and Wendy and Isaac for lots of dinners and games.

Thanks to my "cell"-mates (FPGA humour) Bill Bishop and Dave Grant.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

This dissertation is a study of the interaction among partitioning, scheduling and performance of embedded systems. An embedded system is an element of a larger system, providing a dedicated service to that system [71]. The embedded systems considered in this dissertation are digital electronic systems. Embedded systems are used in many types of applications: in manufacturing for robot control, in consumer products such as cell-phones and in vehicles such as anti-lock breaking on cars or flight-control on airplanes.

A significant subset of embedded systems are real-time embedded systems. For a real-time system to be correct it must be both logically correct and temporally correct. In other words, a real-time system must meet its timing constraints. For example the anti-lock breaking system on a car needs to pump the brakes on the car at the correct time to achieve the correct result, which is a controlled reduction in speed without skidding.

A common implementation platform for embedded systems, both real-time and not, is the System on Chip (SoC). An SoC consists of one or more software programmable processors integrated with application specific hardware on a single integrated circuit. The technique of designing the hardware and software components of an embedded system concurrently is called “hardware/software codesign”. By designing hardware and software concurrently, trade-offs can be made that help to meet system-level design objectives such as timing constraints or power consumption.

1.1 Hardware/Software Codesign

The hardware/software codesign process, or simply codesign process, can be divided into four stages: specification, internal representation, partitioning and synthesis. Ideally, a fifth stage, verification, is integrated into each of the first four stages. The codesign process is reviewed in Chapter 2.

A design is specified which, due to timing constraints or other characteristics, will likely require both software executing on programmable processors and custom hardware encapsulated in coprocessors that communicate with the processors. Partitioning is the act of assigning design components to implementation targets such as programmable processors (software) and coprocessors (hardware). Partitioning may be performed manually or by an automated partitioner. The use of an automated partitioner requires that the specification be transformed into an abstract representation called the internal representation (IR). The IR captures the control and data flow semantics of the design, usually with a graph structure. The IR also has estimates of resources required by the design components. For example: the silicon area if implemented in hardware, the bytes of program memory if implemented in software, or the time required to execute the component in hardware or software.

After partitioning the IR components onto hardware and software implementation targets, the final design is synthesized. The synthesis stage has multiple sub-tasks: interface synthesis, scheduling of software and possibly hardware, and source code generation and compilation.

1.2 Thesis Statement

The focus of this dissertation is the partitioning and scheduling of concurrent real-time systems implemented on SoCs. Some existing codesign systems integrate scheduling into the partitioning stage but only for single-process designs. Those codesign systems that do schedule concurrent systems do so in isolation from the partitioning stage. It is my thesis that partitioning and scheduling of concurrent systems are closely connected and best approached in an integrated manner.

1.3 Dissertation Organization

This dissertation is organized into three studies on the inter-action between partitioning and concurrent scheduling. The first study is on the impact of hardware coprocessors on the feasibility of task sets scheduled by the preemptive Earliest Deadline First (EDF) policy. EDF is an optimal scheduling policy in that it will fail to meet a deadline only if no other policy can meet that deadline. Furthermore EDF can attain, in theory, higher processor utilization than the fixed-priority preemptive policy [67, 50] commonly employed in embedded real-time systems such as VxWorks [105]. EDF has been studied extensively by the real-time scheduling community but little work has been done on EDF by the codesign research community. This may be due to higher run-time scheduling costs for this dynamic priority policy compared to scheduling costs for the fixed-priority policy. Addition of hardware application coprocessors complicates the feasibility analysis for task sets scheduled by EDF policy. In Chapter 3, after reviewing an existing EDF feasibility analysis algorithm, it is extended to integrate the effects of application coprocessors into the analysis.

The second study is a case study of a real-time kernel and application implemented on a System on Programmable Chip (SoPC). The SoPC consists of a soft-core instruction-set processor (or simply processor) and hardware coprocessors implemented on a Field Programmable Gate Array (FPGA). A uniprocessor real-time kernel is developed that schedules by the EDF policy. An automotive application is also developed that consists of multiple tasks scheduled by the kernel. The resulting system has high processor demands and tasks miss deadlines, resulting in incorrect data. Two coprocessors are developed to help meet deadlines: the first implements a subset of the application functionality, and the second implements a subset of the kernel functionality. The impact of these two partitions (processor/coprocessor combination) on the schedule feasibility is evaluated. The case study is presented in Chapter 4.

The third study is on automated partitioners that incorporate schedule feasibility as a partitioning constraint. Furthermore, it is customarily the application that is subject to partitioning but the partitioners that are presented in this study also allow partitioning of the kernel. Two benefits are anticipated from including the kernel in the partitioning:

1. The kernel is invoked at least twice for each task instance: upon task release and task termination. It may also be invoked for inter-task communication or to block on a critical resource. Although the kernel invocations are likely shorter in duration than the task execution, the kernel code's relative frequency of invocation is higher. Therefore a small reduction in kernel time may result in significant application speed-up.

2. Development costs are reduced by code re-use. Moving something from software into hardware requires designing a hardware unit and developing a hardware/software

interface. Since the same kernel can be used for numerous applications, the effort required to implement a part of the kernel in hardware is expended once, but the speed-up may be re-used numerous times.

Two partitioning methods are explored. The first partitioner is a 0-1 non-linear programming (NLP) model which is described in Chapter 5. The NLP model is used to formally define the hardware/software partitioning problem for real-time systems on SoC scheduled by the EDF policy. Solutions to the NLP model are expected to involve long run-times and so a second partitioner is described in Chapter 6 that is an adaptation of the Fiduccia/Mattheyses (FM) circuit partitioning heuristic [31]. The NLP model and FM heuristic adaptation are evaluated in Chapter 7. Case study data and automatically generated problems are used to test the partitioners.

1.4 Contributions

The main contributions described in this dissertation are:

1. incorporation of the preemptive Earliest Deadline First scheduling policy into the hardware/software codesign process,

2. an algorithm to analyze the EDF schedule feasibility of task sets that include tasks which invoke hardware coprocessors,

3. hardware/software partitioning of the application and the kernel, and

4. integration of schedule feasibility into hardware/software partitioning.

These contributions are evaluated in the concluding chapter, Chapter 8.

Chapter 2

Codesign Review

Hardware/software codesign is the practice of designing the hardware and software components of an embedded system concurrently. Codesign takes advantage of the synergism between hardware and software to meet system-level objectives [71]. The codesign flow is comprised of several stages: specification, partitioning, synthesis and verification. The design flow is illustrated in Figure 2.1. The application behaviour and implementation technology is captured in the specification stage. The behaviour is transformed into an internal representation which is usually an abstraction of the information in the specification. The internal representation is used in the partitioning stage to assign application behaviour to implementation technology such as programmable processors and hardware coprocessors. The partitioned application is then synthesized: software and hardware components are compiled and scheduled, and interconnects are generated. Depending on the tool set, some steps may be combined or repeated multiple times. Ideally verification should be integrated into all stages of the design flow.

This review of hardware/software codesign is organized by the stages of the codesign flow: specification (Section 2.1), internal representation (Section 2.2), partitioning (Section

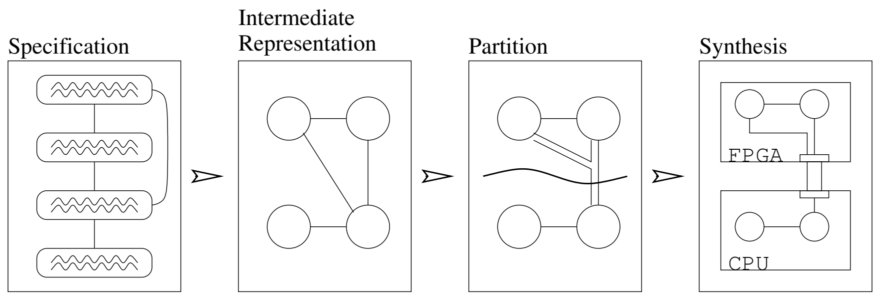


Figure 2.1: Codesign Flow

2.3 and synthesis (Section 2.4. These sections are followed by a discussion of kernel partitioning (Section 2.5), an area closely related to hardware/software codesign. An in-depth review of the Earliest Deadline First scheduling policy is found in Chapter 3.

2.1 Specification

The specification for an embedded system consists of three parts: target technology options, application behaviour description and design constraints [78]. The target technology describes what programmable processors may be used to execute software, to which devices the hardware can be synthesized, and possibly inter-connect alternatives. The hardware target is usually a field programmable gate array (FPGA) or an application specific integrated circuit (ASIC). The software programmable processors are referred to simply as processors. If the processor(s) and application specific hardware are implemented on one integrated circuit (IC), the system is known as a System on Chip (SoC). The design constraints may include ASIC/FPGA size, program and data memory sizes and performance constraints. In the case of a real-time system, timing constraints on the application are specified that must not be violated.

Several methods have been used to specify application behaviour. Executable specifications are often used because they allow early verification of system behaviour via simulation. Both formal specification languages and implementation languages have been used to specify applications. Some formal specification languages that have been used in codesign are Lotos [90], SDL [83], StateCharts [43], and SpecCharts [33]. The implementation languages used for codesign specification can be subdivided into three categories: hardware design languages (HDL), software programming languages and synchronous programming languages.

Hardware Design Languages

- Verilog: used in the Chinook system [26]
- VHDL: used in the Cool system [78]

Software Programming Languages

- C^x: a variant of C, used in the Cosyma system [30]
- HardwareC: another C variant, used in the Vulcan system [39]
- SystemC: a set of C++ class libraries that support hardware constructs, used in Signal Processing Workshop [41]

Synchronous Programming Languages

- Esterel: used in the Polis system [25]

In the CoWare codesign system, Bolsens *et al* [20] use a heterogeneous approach to system specification: rather than use one language to describe all aspects of the system, processes are specified in host language encapsulations (VHDL, Verilog, C, DFL). A language called CoWare describes the inter-connection of the encapsulations. This approach requires the designer to partition the design as part of the specification stage.

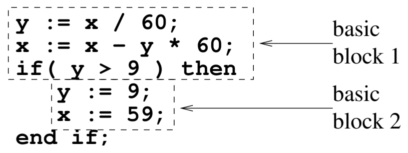
2.2 Internal Representation

With the exception of CoWare, most systems transform the specification into an internal representation (IR). The IR is usually a directed acyclic graph (DAG) with annotations. In some cases, sets of DAGS are used to represent concurrent systems (Vulcan [39] and Dice [47]). Other types of IR are possible: for example the Polis system [25] uses networks of finite state machines, called Codesign FSMs (CFSM).

The DAG consists of nodes and edges. The nodes represent functionality and the edges represent data, control or precedence relations between nodes. The granularity of the nodes varies and the relation represented by the edges varies correspondingly. The node granularity can be subdivided into four categories, as follows.

fine The design is represented at the operator or statement level. Examples of operators are addition, store and branch. A statement is composed of operators, as in "a = b + c". The IR of the Dice system [47] has nodes of fine granularity.

medium-fine The design is represented at the basic block level. A basic block is a sequence of non-branching statements. The following VHDL code consists of 2 basic blocks:



```
y := x / 60;
x := x - y * 60;
if( y > 9 ) then
   y := 9;
   x := 59;
end if;
```

This granularity level is employed in the Cosyma [81] and Lycos [69] systems.

medium-coarse The design is represented at the function (software) or entity (hardware) level. This granularity level is employed by the System Level Intermediate Format

(Slif) [95], and in the Comet [57] and Cool [78] systems. Slif is used in the third study on hardware/software partitioning and is described in detail in Section 2.2.1.

coarse The design is represented at the process level. The CoWare system [20] views the design at this level. This granularity limits the partitioning stage; a stage that CoWare omits. CoWare, instead, focuses on synthesizing inter-process communication.

The node granularity has significant impact on the partitioning stage, described in Section 2.3. Finer granularity allows finer partitioning but greatly increases the solution space. It also impacts the synthesis stage when the partitioned IR is used to generate software code for compilation and hardware code for synthesis, and when interfaces must be synthesized for the partitioned nodes.

2.2.1 System Level Intermediate Format

The System Level Intermediate Format (Slif) is proposed [97] as a standard IR for codesign partitioning. As stated above, Slif represents the design at the function level using a directed acyclic graph. Each node in the graph represents a functional object such as a subroutine or global variable. Each directed edge in the graph represents an access of the destination node by the source node (i.e. invocation of one function by another). The direction of the edge shows the initiator of the access, and so it is called an Access Graph. Nodes and edges are annotated with information as listed in Table 2.1.

The GPP (General Purpose Partitioner) is a freely available package that supports Slif. It provides six benchmark applications from the codesign domain. It also provides a generic Slif generator called "gpslifgen" [97]. Gpslifgen uses statistical data gathered from the six benchmarks. The data includes node fan-out and fan-in based on the node depth (from the root). A correlation was calculated between hardware and software execution

Table 2.1: Slif Node/Edge Annotations

Entity	Annotation	Description
node	size	– gate count (hardware) – instruction count (software)
node	internal computation time (ICT)	– execution time for the node, excluding accesses to other nodes
edge	bit-width	– number of bits transmitted in parallel
	frequency (count)	– average number of accesses over the edge per execution of the edge's source node
node or edge	library component binding	– nodes can be bound to implementations such as FPGAs or processors – edges can be bound to a bus

times and sizes. The number of nodes in the Slif to be generated is dictated by the user. The number of edges are generated automatically and the hardware execution times are chosen randomly based on a distribution. The same is done for sizes. The edge bit-width and frequency values are chosen randomly from an interval derived from the example data. Execution time constraints are also generated automatically.

Gpslifgen was tested by comparing partitioning results on generated examples against partitioning results on the six benchmarks. Eight examples were generated and four partitioning heuristics used. The partitioning results were similar between the generated examples and real benchmarks.

Slif is adopted as the problem description format for the partitioners described in Chapters 5 and 6, and gpslifgen was used to generate test cases in Chapter 7.

2.3 Partitioning

The partitioning stage of the codesign flow is usually referred to as hardware/software partitioning. The problem may be stated as:

> Given an embedded system specification that is implementation independent, assign each component of the system to a target implementation in a manner that optimizes some goal such as speed, power, area, or a combination of these goals. It is assumed that there is at least one hardware target and one software target to which components may be assigned. Multiple components may be assigned to the same target if target resources are sufficient and system design constraints are satisfied.

In the partitioning process, instructions, functions or entire processes are assigned to implementation targets such as processors, FPGAs and ASICs. These heterogeneous targets impact the problem in several ways:

- the unit of node size differs between targets - for processors it may be bytes of code, for ASICs gate count and for FPGAs logic cells or logic elements,
- node execution time varies between targets, and
- edge cost depends on the targets to which its source and destination nodes are assigned.

Hardware/software partitioning is also complicated by potentially contradictory objectives: reduced hardware area and increased speed for example.

Several codesign systems that perform hardware/software partitioning are reviewed in Section 2.3.2. Some of these systems adapt heuristics from the netlist partitioning literature. Netlist partitioning is a technique employed during layout of VLSI (Very Large Scale Integration) circuits. The review of hardware/software partitioning is therefore preceded by a review of relevant netlist partitioning heuristics.

2.3.1 Netlist Partitioning

The circuit partitioning problem arises in VLSI layout. Components called cells (or nodes) are to be laid out in two or more blocks. The cells are connected by wires, called nets. Nets connect two or more cells. Cells may be connected to more than one net. The netlist is the set of all nets to be partitioned. Given a netlist, the problem is to bind the cells to blocks so as to minimize the number of nets spanning the block partitions. Nets that connect cells in different blocks are said to be cut. The set of cut nets is referred to as the cutset. Some solutions impose upper and lower bounds on the number of cells assigned to a block.

The netlist partitioning problem is known to be NP-hard [35]. Therefore algorithmic solutions, which are guaranteed to find an optimal solution, are considered intractable (having run-times too large to be of practical use). Therefore heuristic solutions have been proposed. Heuristic solutions are not guaranteed to find an optimal solution but are intended to find near optimal solutions and have tractable run-times. Two netlist partitioning heuristics are reviewed below: Kernighan/Lin (KL) and Fiduccia/Mattheyses (FM). Note that the heuristics are described using their original symbols which in some cases conflicts with symbols used elsewhere in this thesis.

Kernighan/Lin Node Interchange

Kernighan and Lin published their node interchange heuristic solution to the netlist partitioning problem in 1970 [54]. The KL heuristic (see Algorithm 2.1) partitions a netlist into two blocks and limits the number of cells on each net to two. The node set S is partitioned randomly into subsets A and B. Nodes are successively swapped between A and B until a locally optimal solution is found that minimizes the cutset.

Gain g measures the reduction in cutset when 2 nodes are swapped. For each node

```
Data: node set $S$
start with any arbitrary partition $A, B$ such that $A \cup B = S$, $A \cap B = \emptyset$ ;
all nodes are unlocked ;
repeat
    $k \leftarrow 1$ ;
    repeat
        choose 1 unlocked node from each block to swap, such that gain $g$ is
        maximized ;
        lock swapped nodes ;
        $k \leftarrow k + 1$ ;
    until until all nodes locked;
    choose step $k$ on which the sum of gains $G = \sum_{i=1}^{k} g_i$ is maximized ;
    if $G > 0$ then
        unlock all nodes and restore partition from step $k$ ;
    end
until until $G = 0$;
partition is locally optimal ;
```

Algorithm 2.1: Kernighan/Lin Node Interchange Heuristic

$a \in A$, the external cost E_a (cut edges) is:

$$E_a = \sum_{y \in B} c_{ay}$$

and the internal cost I_a (uncut edges) is:

$$I_a = \sum_{x \in A} c_{ax},$$

where c_{ij} is the weight on the edge between nodes i and j. The difference in external and internal costs for node a is $D_a = E_a - I_a$. The gain if node a from A and node b from B are interchanged is $g = D_a + D_b - 2c_{ab}$. This is the sum of the two nodes D_i's minus the edges that they have in common.

In each step, the gain for all possible swaps is computed which is $O(n^2)$, with $O(n)$ steps per pass (traversal of outer loop) [95]. Thus the complexity is $O(n^3)$ per pass. However it can be modified to run in $O(n^2 \log n)$ time per pass [63]. No lower bound on the number of passes has been proved but has been found experimentally to be a small constant [95].

Fiduccia/Mattheyses Node Move

Fiduccia and Mattheyses [31] improved the Kernighan/Lin heuristic to run in $O(n)$ per pass. This was achieved by moving one cell (node) per step, rather than swapping two cells. Since the number of cells per block changes with each step, upper and lower bounds are specified on the block sizes. Cells are moved only if they do not violate the size bounds. Fiduccia and Mattheyses also generalized the heuristic to partition nets of two or more cells.

More than two cells are allowed to belong to a net. Therefore, cell gain is calculated differently. The first step is to calculate the incidence number $\alpha_A(N)$ of net N on block A. The incidence number is the number of cells on net N that reside in block A (Equation

2.1). C_N is the set of cells on net N.

$$\alpha_A(N) = |\{C | C \in A \quad \text{and} \quad C \in C_N\}| \tag{2.1}$$

Given that $\mathcal{N}_C$ are the nets to which cell C is connected, the gain of moving cell C from block A to block B is:

$$g(C) = |\{N \in \mathcal{N}_C | \alpha_A(N) = 1\}| - |\{N \in \mathcal{N}_C | \alpha_B(N) = 0\}|\,. \tag{2.2}$$

The first part of the expression counts the number of nets to which cell C is connected that have only one cell in block A. Moving cell C to block B will uncut these nets. The second part of the expression counts the number of nets to which cell C is connected that have no cells in block B. Moving cell C to block B will cut these nets. The status of any net to which cell C is connected that has more than one cell in block A and at least one cell in block B will not change with the move of cell C and so does not affect the gain of cell C.

Node Move with Level Gains

Krishnamurthy [58] extended Fiduccia and Mattheyses work by measuring the gain in future steps, not only gain in the current step. For example, if a net has more than two cells in block A, moving one of those cells will not remove the net from the cutset in the current step but makes it possible to remove the net from the cutset in the next step. This concept is called level gains. Sanchis extended Krishnamurthy's heuristic from two block to multiple block partitioning [88].

Hybrid Netlist Partitioning

Areibi and Vannelli [5] use a hybrid genetic algorithm and Tabu search to partition a netlist into multiple blocks. A genetic algorithm [37] is used to generate a number of initial partitions. It should be noted that the name "genetic algorithm" is a misnomer. It is a heuristic, not an algorithm. Therefore these initial partitions are not guaranteed to be optimal. The Tabu search then guides the Sanchis multi-block interchange method to improve the initial partitions. The results indicate that this method produces better results than using just one heuristic.

Of the netlist partitioning heuristics reviewed above, Kernighan/Lin and Fiduccia/Mattheyses were adapted to hardware/software partitioning for codesign as described in Section 2.3.2. The level gain concept of Krishnamurthy and Sanchis is only applicable if nets connect more than two cells. Since hardware/software partitioners work with abstract representations of application behaviour, such as control and data flow graphs (CDFG), rather than electrical circuits, the "connections" between nodes are usually one-to-one, rather than one-to-many.

2.3.2 Hardware/Software Partitioning

The hardware/software partitioning problem differs from the circuit partitioning problem is several ways: the "blocks" (implementation targets) are heterogeneous, the nodes change depending on the block to which they are assigned, and the edges also change depending on the blocks to which their nodes have been assigned. The objective of the partitioning can also be different and multi-faceted. For example: minimize hardware, minimize execution time, minimize power consumption, *etc.* Most of the hardware/software partitioners surveyed below use heuristic solvers because the run-times of algorithmic (optimal) solutions are unacceptably high for use by designers.

The Dice system applies simulated annealing to partition a CDFG [47]. Simulated annealing [56] is a stochastic heuristic (as is the genetic algorithm). The solution is viewed as a material with internal energy that is to be minimized. It starts at a high temperature and the solution moves easily between neighbouring states, even those with higher internal energy. The temperature is slowly lowered (i.e. the material is annealed) and as the temperature decreases, the probability of moving to states with higher energy are reduced. The annealing ends when the temperature reaches zero.

Clustering is applied before simulated annealing to reduce the solution space (the node count is high due to fine granularity). After simulated annealing the Fiduccia/Mattheyses heuristic is applied to the solution (with nodes unclustered) to guide it to a local optimum.

Cosyma [30] also uses simulated annealing to perform partitioning. However it uses a software-oriented approach: all nodes start in software and only enough nodes are moved to hardware to meet timing constraints. The Cosyma partitioner uses estimated execution times [78]. In contrast, Vulcan [39] uses a hardware-oriented approach: all nodes start in hardware and nodes are moved to software as long as performance constraints are satisfied [78]. Vulcan uses a custom iterative heuristic.

Vahid also applies the Fiduccia/Mattheyses heuristic to hardware/software partitioning [95]. Vahid calculates the cost of a partition as a weighted sum of constraint violations such as hardware size, software size, hardware I/O and execution time. Vahid reports experimental results that show simulated annealing finding lower cost partitions 10 out of 16 times. However the simulated annealing showed run-times that exceed $O(n^2)$, whereas FM had run-times that “scale[d] nearly linearly with problem size”.

Kalavade and Lee [52] developed a partitioning heuristic called GCLP (Global Criticality Local Phase). The heuristic attempts to meet performance requirements using a minimum of hardware. GCLP is a constructive heuristic: all nodes start unassigned. Nodes are chosen one at a time for assignment to hardware or software. The objective function is a trade-off between hardware minimization and timing constraints: it is adjusted at each iteration according to the difficulty of meeting timing requirements. Nodes can be swapped after assignment if needed. The schedule is constructed as nodes are assigned (a static schedule of start times). The schedule includes time for hardware-software communication.

The only algorithmic partitioner included in this survey is an integer linear programming (ILP) model used in the Cool system [78]. Nodes are partitioned among multiple software and hardware targets. In addition communication costs between targets are considered. The Cool system uses a second ILP model to perform scheduling. In addition to scheduling nodes, it schedules communication events to avoid bus conflicts. The partitioning and scheduling ILPs are invoked iteratively until a feasible schedule is obtained [78]. No comparison is made with heuristic run-times.

Most of the systems surveyed in this section incorporated nodes costs such as hardware and software size, node execution times in hardware and software and costs for inter-partition edges. Most incorporated timing and area constraints. None of the systems combined scheduling of concurrent systems with the partitioning stage. The only scheduling that was combined with partitioning was that of minimizing the longest path in a CDFG (for a single process). The partitioning result is likely to have a significant impact of the schedule feasibility of concurrent systems. This is considered in the partitioning work of Chapters 5, 6 and 7.

2.4 Synthesis

After partitioning, synthesis is used to transform the abstract graph into an implementation. Synthesis from hardware description languages and software compilation from programming languages are significant parts of the synthesis stage. While these topics are important, they are not unique to hardware/software codesign. The aspects of the synthesis stage that are emphasized here are schedule synthesis (Section 2.4.1) and interface synthesis (Section 2.4.2).

2.4.1 Schedule Synthesis

Some of the partitioners reviewed above combine scheduling with partitioning. However, in most of these cases the scheduling is a static schedule for one process consisting of multiple nodes executed in sequence. This is the type of scheduling is done by the GCLP heuristic [52], the Cool system [78] and Vahid's FM heuristic adaptation [95].

The terminology of scheduling for concurrent systems is discussed here before reviewing codesign systems that schedule such systems. Scheduling is the act of determining which task executes on a processor at a given time. Schedule analysis is determining whether a schedule can meet the deadlines of all the tasks. (A deadline is the time by which a task must finish execution.) Scheduling can be performed off-line (before run-time) or on-line (at run-time). An off-line scheduler may schedule by constructing tables showing the relative order or timing of execution. If a scheduler schedules on-line it determines the next task to execute from a set of ready tasks. It can make this decision based on a static priority, a deadline, or other measure. If a scheduling policy allows one task to be suspended before it is finished to allow another task to execute, then it is a preemptive scheduling policy. Schedule feasibility analysis can also be performed off-line or on-line. If the analysis is performed on-line, it is called dynamic planning and may add significantly to scheduling overhead.

Four codesign systems that construct schedulers are now discussed: Polis, Chinook, CoWare and Vulcan. In the Polis system a codesign finite state machine (CFSM) can be implemented in hardware, software or a peripheral micro-controller [8]. Facilities for communicating events between CFSMs are also synthesized. The generated software is scheduled by either cyclic or fixed-priority policies. Thoen *et al* [93] comment that fixed-priority scheduling is poorly suited to embedded systems because “process priorities have to be used to mimic the timing constraints.... From the designer viewpoint however, these constraints are more naturally specified with respect to the occurrence of observable events.” Furthermore scheduling at the process level is coarse grained and reactive behaviour can be improved with a finer grained scheduling.

In the Vulcan, Chinook and CoWare codesign systems, schedules are constructed for each application. In all cases, threads are formed that consist of serialized operations/nodes from the graph. A run-time scheduler then orders the execution of these threads. Vulcan schedules threads in the order that they become enabled (by arrival of data). Chinook improves on this by scheduling threads out of order to meet timing constraints. Neither Vulcan or Chinook allow interrupts [93] which limits the ability for preemptive scheduling. CoWare allows interrupts for faster reaction to events. It is presumed that scheduling by threads allows finer granularity than scheduling by fixed-priority tasks, thereby improving reaction to events [93]. However no comparisons are made with fixed-priority preemptive schedulers. The synthesized scheduler may have a lower overhead, but would likely be invoked more frequently than a fixed-priority scheduler due to the finer grain of the threads being scheduled. This matter is not pursued further here as it is not the focus of this dissertation.

Another scheduling policy that has not been widely applied in codesign research is the preemptive Earliest Deadline First (EDF) policy. EDF is optimal in the sense that it may fail to meet a deadline only if no other algorithm can meet that deadline. EDF has been extensively studied in the real-time scheduling literature [91]. It can attain, in theory, higher processor utilization than the fixed-priority preemptive policy [67, 50]. EDF also has the desirable quality that timing constraints can be specified with respect to the occurrence of events (see Thoen quote above). EDF schedule analysis can be performed off-line or on-line, although the overhead of on-line EDF schedule analysis discourages its application in embedded systems [9]. Off-line EDF analysis is possible without knowing task arrival times *a priori*, as long as minimum inter-arrival times and worst-case execution times are known. If the EDF schedule analysis is performed off-line, then the run-time EDF scheduler can have overhead comparable to fixed-priority schedulers (tasks are sorted by deadline instead of priority).

EDF schedule analysis is described in detail in Chapter 3, where the impact of hardware coprocessors on schedule feasibility is studied. In Chapters 5 and 6, EDF schedule information is integrated into a partitioning model and heuristic.

2.4.2 Interface Synthesis

Interface synthesis is relevant to this research in that the assumed interface model impacts assumptions and costs used in partitioning and scheduling. For example the GCLP [52] and Cool [78] partitioners associate time costs with edges that connect nodes across the hardware/software interface. The Cool system assumes that a bus is used and assigns hardware costs for connecting to the bus. GCLP does not account for the hardware cost. Communication costs are not explicitly accounted for edges within the same implementation target (i.e. software/software or hardware/hardware).

A bus is also used for the hardware/software communication of systems partitioned in Chapters 5 and 6. The assumptions and costs associated with it are described along with the partitioners.

Table 2.2: STRON-I and UFμK Functions

	STRON-I	UFμK
binary semaphore	$*$	$*$
counting semaphore	$*$	
timer	$*$	$*$
periodic tasks		$*$
fixed-priority preemptive scheduling	$*$	$*$
interrupt handling	$*$	$*$
configurable bus interface		$*$

2.5 Kernel Partitioning

In the codesign flow described above, an application is transformed into a graph representation for partitioning. Then the hardware and software is synthesized, including scheduler synthesis. While it is customarily the application that is subject to partitioning, it is also possible to partition the scheduler. Research into hardware/software partitioning of the kernel (or operating system) is discussed here.

2.5.1 STRON-I

Previous research efforts have investigated implementing real-time kernels almost entirely in hardware. In one such project, the majority of the μITRON kernel functionality was implemented in a coprocessor called STRON-I [77]. STRON-I functions are listed in Table 2.2. A software kernel is required to interact with STRON-I. The software kernel translates system requests into function codes and parameters that are written to STRON-I over the bus. STRON-I communicates with the processor using interrupts and processor-readable registers. As well as translating system requests, the software kernel is required to perform context switches when indicated by STRON-I. The resulting kernel is one third the size of the equivalent software-only kernel.

The number of STRON-I resources (event flags, semaphores, timers) and the number of tasks is fixed at compile time. A prototype was implemented on a XC4010 Xilinx FPGA with 3 tasks, 3 event flags, 3 semaphores and 3 external interrupts, with a resulting size of 4300 gates and speed of 12 MHz. The coprocessor size scaled linearly with the number of tasks and resources. The kernel functions implemented in hardware were shown to be 6 to 50 times faster than the equivalent functions in software.

2.5.2 UFμK

The FASTCHART project [66] mated a custom processor that could perform a context switch in one cycle with a kernel coprocessor called the RTU (Real Time Unit). The RTU shares several features with STRON-I including the communication method, priority preemptive scheduling and a number of kernel functions. In order to make the hardware kernel applicable to a broader range of applications, the RTU was extracted from FASTCHART so that it could be implemented as an ASIC. The RTU could then be interfaced with various real-time system buses, such as the VME bus[2]. A key difference with STRON-I is that the RTU can support multiple processors.

The RTU has been commercialized as the UltraFast Micro Kernel (UFμK) [84]. UFμK functions are listed in Table 2.2. UFμK is supported on several FPGAs, allowing the number of tasks and resources to be configurable.

2.5.3 δ Framework

Both the STRON-I and UFμK implement the majority of kernel functions in a hardware coprocessor. A finer-grained partitioning of the kernel is investigated with the δ SoC/RTOS Codesign Framework [72]. The user can choose from a list of strategic parts of the Atalanta kernel to move into hardware. The list currently includes:

1. a lock cache for synchronization operations,
2. a deadlock detection unit for multi-processor systems and
3. a dynamic memory manager.

The goal is to achieve significant application speed-up using a small amount of hardware.

In [62], the δ Framework was used to compare three kernel configurations:

1. software (Atalanta)
2. hardware/software (Atalanta with hardware lock cache)
3. hardware (RTU)

The 3 configurations were simulated to implement a multi-processor data-base application which required many task synchronizations. The hardware/software configuration had speed-ups of 19-41% over the software, while the hardware configuration had speed-ups of 36-50% over the software. Synthesis of the lock cache required 7435 gates, while the RTU took approximately 250000 gates.

2.5.4 SSCoP

The Spring operating system [22] supports distributed real-time systems made up of multi-processor nodes. It uses dynamic-planning scheduling to admit and schedule newly arrived tasks. This approach constructs a plan for task executions. It attempts to add a new task to the plan such that its deadline is met and all of the previously scheduled tasks still meet their deadlines. If no feasible plan is found, then the new task is not admitted.

Each multiprocessor Spring node has application processors and system processors. The application code executes on the application processors and the system processors perform scheduling and operating system tasks. The Spring scheduling coprocessor (SSCoP) is a VLSI implementation of the Spring scheduling algorithm and works with the system processors. The algorithm schedules non-preemptable tasks that have deadlines, resource requirements and precedence constraints. Multiple scheduling policies are supported, including Earliest Deadline First (EDF). The size of SSCoP varies with the number of tasks and resources supported and the word size. For example, a SSCoP that supports 64 tasks, 32 resources and a word size of 32 bits uses 165492 gates.

The system processor must write all task information into memory-mapped SSCoP registers. It must also perform pre-processing and post-processing for any newly arrived tasks, as well as waiting for SSCoP to construct a new plan. By considering system processor pre-processing and post-processing times as well as SSCoP execution times, a SSCoP which can schedule 64 tasks speeds up scheduling by a factor of 4.2 – 6.5 over an all-software solution. This speed up factor is calculated for a SSCoP clocked at 100MHz and a system processor (M68020) capable of 1 MIPS (millions of instructions per second).

A case study is presented in Chapter 4 in which both an application and its kernel are manually partitioned. The kernel performs EDF scheduling of preemptable tasks (without dynamic-planning).

2.6 Summary

A codesign flow of four stages has been presented: specification, internal representation, partitioning and synthesis. Examples from codesign literature have been presented for each of these stages. The lack of integration between partitioning and scheduling of concurrent systems was noted. Partitioning of the kernel has also been introduced, in addition to partitioning of the application. The Earliest Deadline First scheduling policy is reviewed further in Chapter 3.

Chapter 3

EDF Scheduling of Embedded Systems

The Earliest Deadline First (EDF) scheduling policy is an important real-time scheduling policy. It assigns task priorities based on their deadlines and it is optimal in the sense that it will fail to meet a deadline only if no other policy can meet that deadline. EDF has been extensively studied by the real-time scheduling community but little work has been done on EDF by the codesign research community, due in part to the run-time scheduling overhead [9]. However, the overhead is not excessive ($O(logn)$) and this policy makes possible higher processor utilization [67]. Most of the codesign systems reviewed in Chapter 2 that schedule concurrent software do so by synthesizing threads of execution off-line. The exception, Polis, uses fixed-priority preemptive scheduling.

EDF is a real-time scheduling policy that has the potential to increase processor utilization [50] but has received little attention from the codesign community. Therefore EDF has been selected as the scheduling policy for this investigation into partitioning and scheduling of embedded systems. In Section 3.1, an introduction into the theory of EDF scheduling is given. In Section 3.2, an off-line EDF feasibility algorithm is extended to accommodate task sets that block on coprocessors during their execution.

3.1 EDF Scheduling Summary

Under the preemptive EDF policy, of all ready tasks, the task with the earliest deadline is executed first. If another task arrives with an earlier deadline, it will preempt the currently executing task. All discussion of the EDF policy in this thesis refers to preemptive EDF. In order to describe the feasibility analysis of systems scheduled by EDF, some terminology needs to be introduced. Periodic tasks occur at regular intervals. Periodic task τ_i has start time s_i, period T_i, relative deadline D_i and worst-case execution time C_i. τ_i is released at the start of each period at time $r_i = s_i + mT_i, m = 0, 1, \ldots$. The task must complete by (absolute) deadline $d_i = r_i + D_i$. Aperiodic tasks are released in response to an event. Aperiodic task τ_i does not have a start time or a period: it has relative deadline D_i and worst-case execution time C_i.

Figure 3.1 shows two tasks: τ_1 is periodic and τ_2 is aperiodic. Task releases are indicated with an up arrow and task deadlines are indicated with a down arrow. τ_1 has start time $s_1 = 2$, relative deadline $D_1 = 8$, period $T_1 = 12$ and worst-case execution time $C_1 = 4$. It has release dates $r_1 = 2, 14, \ldots$ and corresponding deadlines $d_1 = 10, 22, \ldots$. τ_2 is released aperiodically at times $t = 0, 16, 21$. At $t = 16$ τ_2 preempts τ_1 because its deadline is closer. τ_1 is resumed at $t = 18$ after τ_2 terminates.

The purpose of EDF schedule analysis is, given task set τ, determine whether all tasks in the set can be scheduled by the preemptive EDF policy such that no task misses its deadline. The analysis is for periodic tasks, but can include a sporadic task τ_i that has a minimum inter-arrival time T_i which can be used as a period in the analysis. Task sets

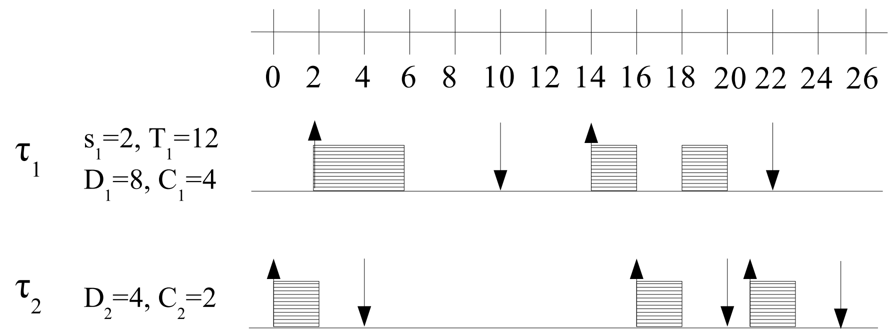


Figure 3.1: Task Notation

with both periodic and sporadic tasks are denoted *hybrid* task sets. If task deadlines are allowed to be shorter (or longer) than their periods, then it is a *generic* hybrid task set. In their book [91], Stankovic *et al* develop a feasibility analysis for generic hybrid task sets. The analysis is based on processor demand and is explained below.

A necessary but not sufficient condition for the feasibility of a generic hybrid task set scheduled under EDF is that the processor utilization U (defined by Liu and Layland [67]) is bounded above by 1:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{3.1}$$

U sums the fraction of processor time required per task.

The rest of the analysis is based on the most constraining scenario where all task releases are synchronous ($\forall i$: $s_i = 0$). Unless specified otherwise, references to the task set assume this synchronous task set. After checking that $U \leq 1$ the algorithm checks processor demand. Processor demand, $h(t)$, in the interval $[0, t)$ is defined as:

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right) C_i. \tag{3.2}$$

Processor demand at time t measures the work required by all tasks having deadlines in

$[0, t]$. Processor demand is applied in the following theorem:

Theorem 3.1 (Processor Demand Condition [91]). *Any given [generic] hybrid task set is feasible under EDF scheduling* if *and* only if

$$\forall t, h(t) \leq t.$$

However, it is not necessary to test processor demand on all intervals in $[0, t)$; it is sufficient to test only when task deadlines occur. Also, an upper bound on the interval over which $h(t)$ is checked can be determined. Three upper bounds are calculated in [91] and the least upper bound is chosen to define the interval. The three upper bounds are:

1. $ub_1 = \max\left\{D_{\max}, \frac{\sum_{i=1}^{n}(1-\frac{D_i}{T_i})C_i}{1-U}\right\}$,
2. $ub_2 = \frac{\sum_{D_i \leq T_i}(1-\frac{D_i}{T_i})C_i}{1-U}$, and
3. $L =$ synchronous busy period.

The derivation of upper bounds ub_1 and ub_2 is explained in Appendix A. It is the third upper bound L that is used in extending the analysis in Section 3.2. Therefore L is explained here. The synchronous busy period, L, is based on this theorem (recall that all tasks are assumed to start at $t = 0$):

Theorem 3.2 (Liu and Layland [67]). *When the deadline driven scheduling algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to an overflow.*

The interval of time preceding the first idle period is called the *synchronous busy period.* Note that the idle period may have duration equal to 0; in essence, at the end of the synchronous busy period no work has been previously released that still requires execution.

Stankovic *et at* describe the following iterative method for calculating the duration of the synchronous busy period, L.

Apply recursively until $L^{m+1} = L^m$:

$$\begin{cases} L^{(0)} &= \sum_{i=1}^{n} C_i, \\ L^{(m+1)} &= W(L^{(m)}), \end{cases} \tag{3.3}$$

where

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i. \tag{3.4}$$

Since all tasks are released synchronously, $L^{(0)}$ is the sum of task execution times. During $L^{(0)}$, other work may be released which is added to L. This is repeated until there is no change to L. Computation of iterative Equation 3.3 takes $O(n \sum_{i=1}^{n} C_i)$ time, if $U < 1$ [91].

Stankovic *et al*'s algorithm for feasibility analysis [91] can be summarized by the following four steps. Algorithm 3.1 lists the corresponding pseudocode. Note that the upper bounds ub_1 and ub_2 are omitted in this discussion of the algorithm.

1. check that $U \leq 1$
2. determine upper bound, L, for test period
3. define set S of deadline events in $[0, L)$
4. for each event v in S, check that $h(t) < t$

3.2 EDF Scheduling with Coprocessors

Task sets that block on coprocessors present a unique challenge to the EDF feasibility analysis. A task that blocks on a coprocessor is similar to a task that blocks on a critical

```
**Data**: task set $\tau$
**Result**: schedule feasibility
**if** $U > 1$ **then** **return** *"not feasible"* ;
$S = \cup_{i=1}^{n}\{mT_i + D_i : m = 0, 1, \ldots\} = \{v_1, v_2, \ldots\}$ ;
$k \leftarrow 1$ ;
**while** $v_k < L$ **do**
    **if** $h(v_k) > v_k$ **then** **return** *"not feasible"* ;
    $k \leftarrow k + 1$ ;
**end**
**return** *"feasible"* ;
```

Algorithm 3.1: EDF Feasibility Analysis

section. EDF feasibility analysis of tasks blocked in critical sections has been studied [7, 24, 91]. There is however a difference between tasks blocked on coprocessors and tasks blocked on critical sections. Tasks blocked on a critical section are waiting for another task to finish in the critical section. If the task is blocked, there is another task that can be executed. When a task blocks on a coprocessor, there may or may not be another task ready to execute. This can introduce additional idle times into the schedule which can impact the task's ability to meet deadlines. The feasibility analysis for task sets which block on coprocessors is developed below.

When part of a task is moved from the processor to a coprocessor, it contributes to schedule feasibility in two ways. First, it should reduce total task execution time (i.e. execution time for τ_i on processor and coprocessor) which helps τ_i to meet its deadline. Second, it reduces processor utilization which increases feasibility of the task set. In this chapter, a method is proposed to incorporate both benefits into the feasibility analysis of task sets with coprocessors.

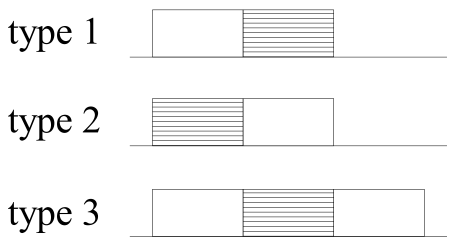


Figure 3.2: Coprocessor Task Type

As illustrated in Figure 3.2, there are three types of coprocessor use by a task. The clear box represents execution on the processor and the shaded box execution on the coprocessor. The first and second types do not require any change to the feasibility analysis of Algorithm 3.1.

type 1 The software portion of the first type can be represented by a periodic or sporadic task. The only change would be to subtract the worst-case execution time of the coprocessor, C_{coproc}, from the software task deadline: $D'_i = D_i - C_{\text{coproc}}$. This would ensure that the coprocessor finishes by deadline D_i. (It is assumed that the coprocessor is available immediately.)

type 2 The software portion of the second type can be represented by a sporadic task with minimum inter-arrival rate as determined by the coprocessor. Again, the deadline is shortened: $D'_i = D_i - C_{\text{coproc}}$.

type 3 The third type, tasks that block on coprocessors (called "coprocessor-blocked tasks"), requires further analysis.

To understand the approach taken to addressing this problem, consider Figure 3.3. The first line shows the task implemented completely in software. The second line shows

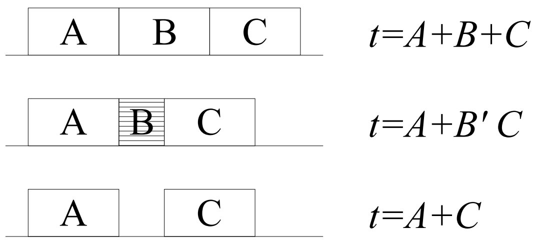


Figure 3.3: Coprocessor Task Analysis

Part B of the task implemented on a coprocessor which has decreased the overall task execution time. The third line omits Part B of the task because it no longer executes on the processor. The total execution time of the second line can be used as the task execution time thereby incorporating some of the coprocessor benefit (reduced total execution time for the task). A less conservative approach is to use the execution times of the third line as the task execution time, incorporating the full benefit of the coprocessor (reduced total execution time and parallel execution). Both approaches are discussed in Section 3.2.1 with the emphasis being placed on the less conservative approach.

The EDF feasibility analysis for coprocessor-blocked tasks is developed first for one task blocking on a coprocessor once per task instance, in Section 3.2.1. The analysis is extended to one task blocking on coprocessors multiple times per task instance in Section 3.2.2, and the impact on schedule feasibility is quantified in Section 3.2.4.

3.2.1 One Task, Blocking on Coprocessor Once

In this section, the feasibility analysis is developed for task sets in which one task blocks on a coprocessor, and only blocks on the coprocessor once per task instance. Since coprocessor-blocked tasks have three execution times (on processor before coprocessor, on coprocessor, and on processor after coprocessor) worst-case execution times are represented in vectors, $\vec{C}_i$. In the case of a regular task (no coprocessor) the size of the vector is $|\vec{C}_i| = 1$. For a task with one coprocessor block per task instance, $|\vec{C}_i| = 3$. The three elements of its vector are:

$\vec{C}_{i,1}$	first execution on processor,
$\vec{C}_{i,2}$	execution on coprocessor, and
$\vec{C}_{i,3}$	second execution on processor.

A simple approach to the feasibility analysis would be to use the analysis of Algorithm 3.1 using the vector sum as the worst-case execution time for a task (this is summarized in Algorithm 3.2). While this does account for any reduction in total task execution time, it does not take advantage of the fact that a second task can use the processor while the first task is waiting for the coprocessor to finish. Stated another way: some task-level parallelism is ignored. (Tasks which employ coprocessors are actually a highly constrained form of multi-processing.)

```
Data: task set $\tau$
Result: schedule feasibility
foreach $\tau_i$ do
    $C_i = \sum_{a=1..|\vec{C}_i|} \vec{C}_{i,a}$ ;
end
invoke Algorithm 3.1 ;
```

Algorithm 3.2: Simple Feasibility with Coprocessors

The approach taken in developing the analysis below is to represent the coprocessor-blocked task, τ_i, as a processor with three ordered subtasks: $\tau_{i,1}$, $\tau_{i,2}$ and $\tau_{i,3}$. Given τ_i with parameters T_i, D_i and $|\vec{C}_i| = 3$, three subtasks are derived as listed in Table 3.1. The last subtask in the order, $\tau_{i,3}$, has deadline $D_{i,3} = D_i$. The deadline of $\tau_{i,2}$ is modified to ensure that $\tau_{i,2}$ finishes with enough time for $\tau_{i,3}$ to complete. The same type of modification is applied to the deadline of $\tau_{i,1}$. By modifying the deadlines in this manner, the correct order of execution is enforced since the task with the earlier deadline is executed first. This approach is the same type of deadline modification employed by Stankovic *et al* in Figure 7.5 of [91]. The impact of this deadline modification scheme on the feasibility analysis is discussed in Section 3.2.4.

Table 3.1: Coprocessor Task Division

Task	Target	Period	Deadline	Worst-Case Exec. Time
$\tau_{i,1}$	processor	$T_{i,1} = T_i$	$D_{i,1} = D_i - \vec{C}_{i,2} - \vec{C}_{i,3}$	$C_{i,1}$
$\tau_{i,2}$	coprocessor	$T_{i,2} = T_i$	$D_{i,2} = D_i - \vec{C}_{i,3}$	$C_{i,2}$
$\tau_{i,3}$	processor	$T_{i,3} = T_i$	$D_{i,3} = D_i$	$C_{i,3}$

τ_i is replaced by $\tau_{i,1}$ and $\tau_{i,3}$ in the task set that is analyzed using the processor demand analysis of Algorithm 3.1. $\tau_{i,2}$ is not in the task set since it executes on the coprocessor, not the processor. The short-coming of this analysis is that it ignores the required time lapse ($\vec{C}_{i,2}$) that must occur between $\tau_{i,1}$ and $\tau_{i,3}$. Consider the example in Figure 3.4. The coprocessor-blocked task, τ_1, would be replaced in the task set by $\tau_{1,1}$ and $\tau_{1,3}$. $\tau_{1,2}$ is excluded from the task set. The deadline of $\tau_{1,1}$ is

$$\begin{aligned} D_{1,1} &= D_1 - \vec{C}_{1,2} - \vec{C}_{1,3} \\ &= 11 - 4 - 5 \\ &= 2. \end{aligned}$$

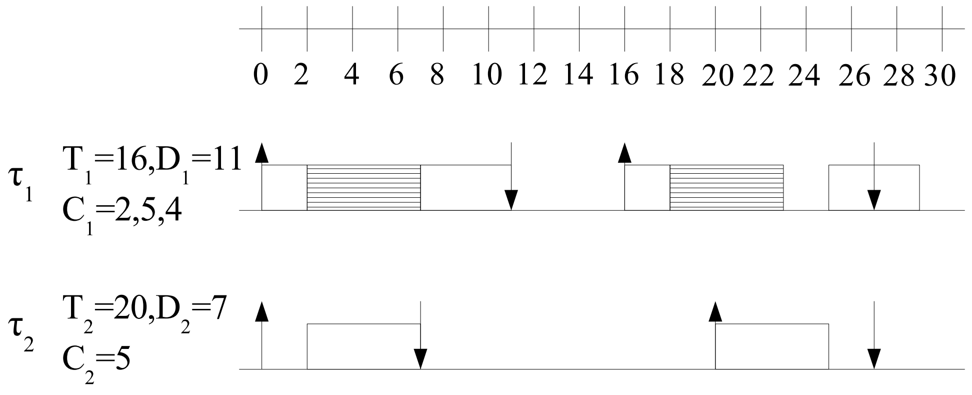


Figure 3.4: Feasibility Failure

Following Algorithm 3.1, $U = 0.625$, $L = 11$ and $S = \{2, 7, 11\}$. At each deadline event in S, there is enough time to meet the processor demand ($\forall v \in S : h(v) \leq v$). The modified task set therefore passes the analysis. However a deadline is missed at $t = 27$.

The feasibility analysis assumes that $\tau_{1,3}$ can execute at any time after its release. However it cannot execute until after the coprocessor ($\tau_{1,2}$) has finished, creating an idle period on the processor. It is this coprocessor-induced idle that causes the missed deadline at $t = 27$. The goal of the analysis developed below is, given a task set, to determine whether a coprocessor-induced idle can result in a missed deadline. This new analysis is done in addition to the analysis of Algorithm 3.1.

Consider the example given in Figure 3.5. τ_1 is a coprocessor-blocked task. In this scenario, the processor is idle while $\tau_{1,2}$ executes on the coprocessor. This implies that all other tasks have finished execution and are not ready to be released before the end of the coprocessor idle. This scenario is the basis for the following lemma:

Lemma 3.1. *Given a task set τ in which one task τ_x blocks on a coprocessor once, with logical subtasks $\{\tau_{x,1}, \tau_{x,2}, \tau_{x,3}\}$, the coprocessor-induced idle has the worst effect on the*

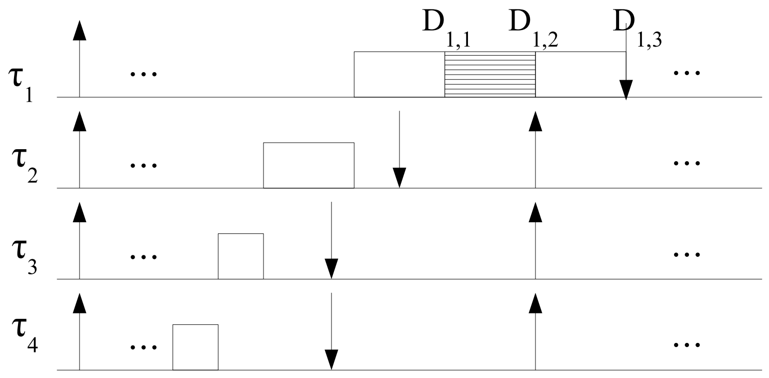


Figure 3.5: Coprocessor-Induced Idle Period

schedule when:

1. *$\tau_{x,3}$ has no slack, and*

2. *all other tasks are released synchronously at the end of the coprocessor-induced idle (i.e. at $t = D_{x,2}$).*

Before starting the proof of Lemma 3.1, the concept of loading factor needs to be introduced. Loading factor on the interval $[t_1, t_2)$ is defined as

$$u_{[t_1,t_2)} = \frac{h_{[t_1,t_2)}}{(t_2 - t_1)}.$$

$h_{[t_1,t_2)}$ is the sum of work released not earlier than t_1 with deadline not later than t_2:

$$h_{[t_1,t_2)} = \sum_{t_1 \le r_k, d_k \le t_2} C_k,$$

where r_k, d_k and C_k are release, deadline and worst-case execution time for jobs (task instances) in that interval. Loading factor is the fraction of the interval required to meet its processor demand. The absolute loading factor is the maximum loading factor of all possible intervals:

$$u = \sup_{0 \leq t_1 < t_2} u_{[t_1,t_2)}.$$

The proof of Lemma 3.1 now follows.

Proof. The worst-case schedule feasibility occurs when:

- **Part 1:** $\tau_{x,3}$ has no slack ($D_{x,3} = \vec{C}_{x,3}$)

  - If $\tau_{x,3}$ has no slack, then no other work can be done before $D_{x,3}$. (i.e. if there exists another task with deadlines before $D_{x,3}$, then at $t = D_{x,3}$, $h(t) > t$ which makes the schedule infeasible.)
  - Adding slack S to $\tau_{x,3}$ would allow S units of work to be done before the revised deadline $D'_{x,3} = \vec{C}_{x,3} + S$. If there exists other tasks with deadlines before $D'_{x,3}$ and whose total work does not exceed S, then processor demand at $D'_{x,3}$ will not be exceeded. Any tasks with deadlines greater than $D'_{x,3}$ are not affected since the work required by $\tau_{x,3}$ is not changed. Therefore adding slack to the deadline of $\tau_{x,3}$ will either not impact schedule feasibility or if it does, it will make those tasks with deadlines before $D'_{x,3}$ feasible.

- **Part 2:** all other tasks are released synchronously with $\tau_{x,3}$
  The remaining part of this proof is the same, in essence, as the proof of Lemma 3.1 in [91] except for modifications to the coprocessor-blocked task as shown below.

  Let τ be the set of all tasks (without the assumption of synchronous release) and τ' the corresponding synchronous task set and let the coprocessor-blocked task, τ_x, be excluded from τ and τ'. When the processor demands, $h_{[t_1,t_2)}$ and $h'_{[t_1,t_2)}$, of τ and τ' are computed, they are augmented by the coprocessor-blocked task as follows:

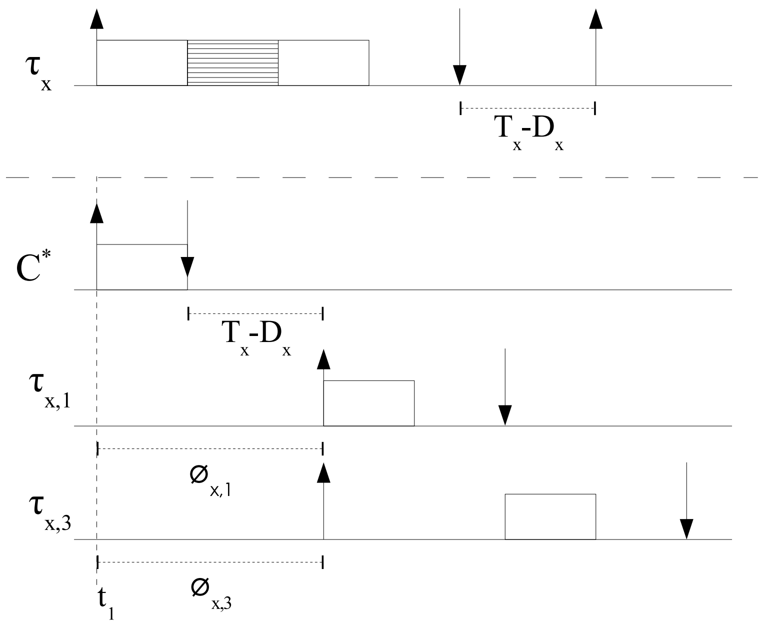


Figure 3.6: C^* and τ_x phases

– In calculating the processor demand, a unit of work C^* is released at t_1 with duration $C^* = \vec{C}_{x,3}$, representing the part of the coprocessor-blocked task that must execute after the coprocessor induced idle. The deadline of C^* is D^* which is equal to $\vec{C}_{x,3}$ (no slack). Furthermore, τ_x is replaced by $\tau_{x,1}$ and $\tau_{x,3}$. Since C^* represents the remaining work of the previous instance of τ_x which has a deadline at $t = \vec{C}_{x,3}$, the next instance of τ_x is released at $T_x - D_x + \vec{C}_{x,3}$ after t_1. Tasks $\tau_{x,1}$ and $\tau_{x,3}$ are said to have phases $\phi_{x,1} = \phi_{x,3} = T_x - D_x + \vec{C}_{x,3}$. Only a coprocessor-blocked task is said to have phase (note that phase and start time are not related). Figure 3.6 shows relationship between τ_x and C^* and $\tau_{x,1}$ and $\tau_{x,3}$ with phases.

In order to prove that synchronous release of all other tasks but the coprocessor-blocked task is the worst-case scenario, it is sufficient to prove that the absolute loading factor ("loading factor") u of τ is bounded above by the loading factor u' of τ'. It is therefore sufficient to prove that $\forall[t_1, t_2)$ there exists an interval $[t'_1, t'_2)$ such that $h_{[t_1,t_2)} \leq h'_{[t'_1,t'_2)}$, where the coprocessor-blocked task is included as described in the previous paragraph. For any interval $[t_1, t_2)$, the processor demand for τ is bounded above by:

$$h_{[t_1,t_2)} \leq C^* + \sum_{D_i+\phi_i \leq t_2-t_1} \left(1 + \lfloor \frac{t_2 - t_1 - (D_i + \phi_i)}{T_i} \rfloor \right) C_i.$$

Addition of ϕ_i to this equation only affects $\tau_{x,1}$ and $\tau_{x,3}$.

Now consider the synchronous case, τ', and let $t'_1 = 0$ and $t'_2 = t_2 - t_1$. Processor demand is equal to:

$$\begin{aligned} h'_{[t'_1,t'_2)} &= C^* + \sum_{D_i+\phi_i \leq t'_2-t'_1} \left(1 + \lfloor \frac{t'_2 - t'_1 - (D_i + \phi_i)}{T_i} \rfloor \right) C_i \\ &= C^* + \sum_{D_i+\phi_i \leq t_2-t_1} \left(1 + \lfloor \frac{t_2 - t_1 - (D_i + \phi_i)}{T_i} \rfloor \right) C_i. \end{aligned}$$

Processor demand of the synchronous case, $h'_{[t'_1,t'_2)}$, equals the upper bound of the other case, $h_{[t_1,t_2)}$. Therefore:

$$h_{[t_1,t_2)} \leq h'_{[t'_1,t'_2)}.$$

□

The analysis that is performed is similar to Algorithm 3.1 except that the coprocessor-blocked task has a phase. Modifications are required to the synchronous busy period calculation and processor demand calculation. The phase of the coprocessor-blocked task is redefined here to simplify calculations. In the proof above, the portion of the coprocessor-blocked task remaining after the coprocessor-induced idle was represented by C^* and the

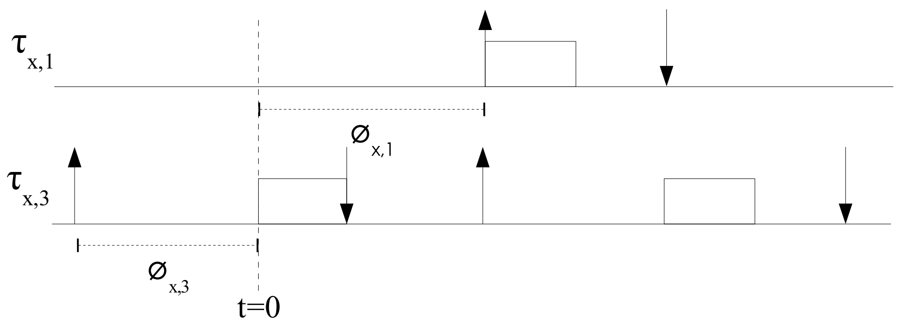


Figure 3.7: $\phi_{x,1}$ and $\phi_{x,3}$

next instance of the task had phase $\phi_{x,1} = \phi_{x,3} = T_x - D_x + C_{x,3}$. This can also be represented by modifying the phase of $\tau_{x,3}$: $\phi_{x,3} = -(D_x - C_{x,3})$. By doing this $\tau_{x,3}$ has an initial deadline at $t = C_{x,3}$. Also note that $D_x - C_{x,3} = D_{x,2}$. The phases of the two subtasks of the coprocessor-blocked task τ_x become:

$$\begin{aligned} \phi_{x,1} &= T_x - D_{x,2}, \\ \phi_{x,3} &= -D_{x,2}, \end{aligned}$$

and every other task has phase $\phi_i = 0$. The change in $\phi_{x,3}$ can be seen by comparing Figures 3.6 and 3.7. In Figure 3.7, C^* has been incorporated into $\tau_{x,3}$.

The synchronous busy period is now calculated as follows:

Apply recursively until $L'^{\,m+1} = L'^{\,m}$:

$$\begin{cases} L'^{(0)} = \sum_{\phi_i \le 0} C_i, \\ L'^{(m+1)} = W'(L'^{(m)}), \end{cases} \tag{3.5}$$

where

$$W'(t) = \sum_{\phi_i \le t} \left\lceil \frac{t - \phi_i}{T_i} \right\rceil C_i. \tag{3.6}$$

When $\phi_{x,1} = \phi_{x,3} = 0$, then $L' = L$. When $\phi_{x,1} = T - D_{x,2}$ and $\phi_{x,3} = -D_{x,2}$, then the result is different: at $t = 0$ all of the regular non-coprocessor tasks are released as well as coprocessor subtask $\tau_{x,3}$. The sum of their worst-case execution times forms $L'^{(0)}$. As in calculating L, any work released in $L'(0)$ is added to L' repeatedly until their is no new work to add. Note that each task in the task set has worst-case execution time vectors of size one and so it is referred to simply as C_i. The processor demand is also modified for phase:

$$h'(t) = \sum_{D_i+\phi_i \leq t} \left(1 + \left\lfloor \frac{t - (D_i + \phi_i)}{T_i} \right\rfloor\right) C_i. \tag{3.7}$$

This modification includes $\tau_{x,3}$ for $t \geq C_{x,3}$ and includes $\tau_{x,1}$ for $t \geq D_{x,3} + \phi_{x,3}$.

Algorithm 3.3 details the feasibility analysis of a task set with one coprocessor task that blocks on the coprocessor once per task instance. Note that construction of event set S has also been modified for phase shifts. The analysis is done twice: the first pass is the same as Algorithm 3.1 - all phases are equal to zero, the second pass is done with coprocessor task phases as discussed above.

An example is introduced in Figure 3.8 to demonstrate Algorithm 3.3. There are four tasks in the task set, one of which (τ_1) blocks on a coprocessor. τ_1 is represented by subtasks $\tau_{1,1}$, $\tau_{1,2}$ and $\tau_{1,3}$ ($\tau_{1,2}$ is not included in the EDF analysis). The deadline of $\tau_{1,1}$ is indicated by a dashed down arrow. First processor utilization is checked:

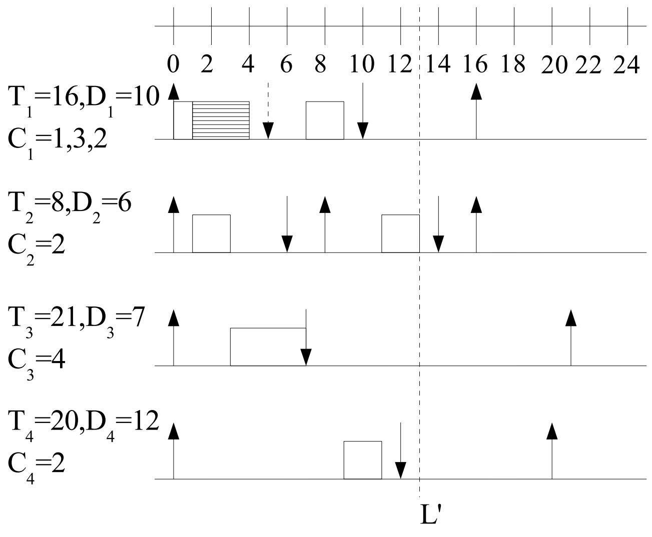
$$U = 0.7280 \leq 1.$$

The first pass of the loop (all phases equal zero) is shown in Figure 3.8(a). $L' = 13$ and $S' = \{5, 6, 7, 10, 12\}$. At each event in S', there is enough time to meet the processor demand ($\forall v \in S' : h'(v) \leq v$).

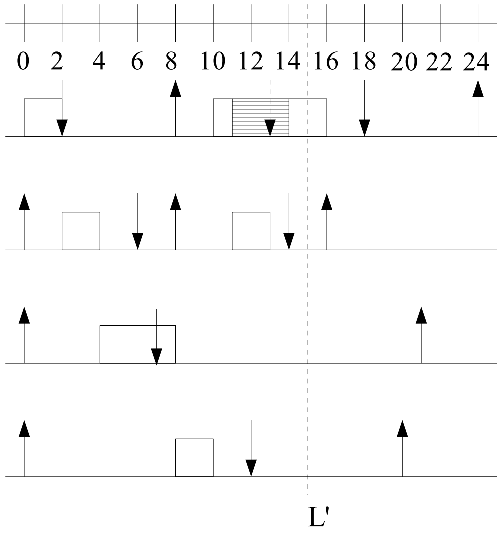
The second pass of the loop is shown in Figure 3.8(b). The τ_1 subtask phases are set to $\phi_{1,3} = -D_{1,2} = -8$ and $\phi_{1,1} = 16 - 8 = 8$, while the other task phases remain zero.

```
**Data**: task set $\tau$ (coprocessor task $\tau_i$ replaced by $\tau_{i,1}$ and $\tau_{i,3}$)
**Result**: schedule feasibility
if $U > 1$ then return "not feasible" ;
loop twice
    for $\tau_i \in \tau$ do
        if $|\vec{C}_i| = 1$ then
            $\phi_i = 0$ ;
        else
            /*set coprocessor subtask phases */
            if first iteration then
                $\phi_{i,1} = \phi_{i,3} = 0$ ;
            else
                $\phi_{i,1} = T_i - D_{i,2}$ ;
                $\phi_{i,3} = -D_{i,2}$ ;
            end
        end
    end
    $S' = \cup_{i=1}^{n}\{mT_i + \phi_i + D_i : m = 0, 1, \ldots\} = \{v_1, v_2, \ldots\}$ ;
    $k \leftarrow 1$ ;
    while $v_k < L'$ do
        if $h'(v_k) > v_k$ then return "not feasible" ;
        $k \leftarrow k + 1$ ;
    end
endloop
return "feasible" ;
```

Algorithm 3.3: Feasibility: One Task, Blocked Once



(a) $\forall \tau_i : \phi_i = 0$



(b) $\phi_{1,1} = 8, \ \phi_{1,3} = -8$

Figure 3.8: Example: One Task, Blocked Once

$L' = 15$ and $S' = \{2, 6, 7, 12, 13, 14\}$. Analyzing the processor demand at each deadline event, $h'(2) = 2,\ h'(6) = 4,\ h'(7) = 8, \ldots$. At $t = 7$, the processor demand exceeded the available time and so the analysis concludes that this task set cannot be feasibly scheduled by the preemptive EDF policy.

In both passes of Algorithm 3.3, processor demand is checked over the interval $[0, L')$. In the first pass L' is calculated with all phases zero. In the second pass L' is calculated using $\phi_{x,1}$ and $\phi_{x,3}$ as described above. In both cases, the calculation of L' ignores the real possibility that a coprocessor-induced idle could occur. Therefore it is necessary to consider the effect of such an event on each pass of the algorithm. If a coprocessor-induced idle arises during $[0, L')$ of the first pass, it implies that all other tasks have completed and only coprocessor-blocked subtask $\tau_{i,3}$ remains to be executed (i.e. a coprocessor-induced idle has occurred). This is in fact the case that the second pass of the algorithm checks. If a coprocessor-induced idle arises during $[0, L')$ of the second pass, it cannot result in any worse scenario than was started with for the second pass. So the occurrence of a coprocessor-induced idle during $[0, L')$ of either pass does not invalidate the analysis.

3.2.2 One Task, Blocking on Coprocessors Multiple Times

Algorithm 3.3 is now extended to one coprocessor task blocking on coprocessors multiple times per task instance. Let coprocessor task, τ_i, have worst-case execution time vector of size $N = |\vec{C}_i|$. $N = 3, 5, 7, \ldots$ because each coprocessor block is preceded and succeeded by processor execution ($N = 1$ implies a regular task). All odd-numbered elements of $\vec{C}_i$ represent execution on the processor and all even-number elements represent execution on the coprocessor. The analysis of Algorithm 3.3 is done once with all phases equal zero and then is repeated once per coprocessor block with phases as indicated in Figure 3.9. The coprocessor task is not shown in it's entirety, only the part corresponding to

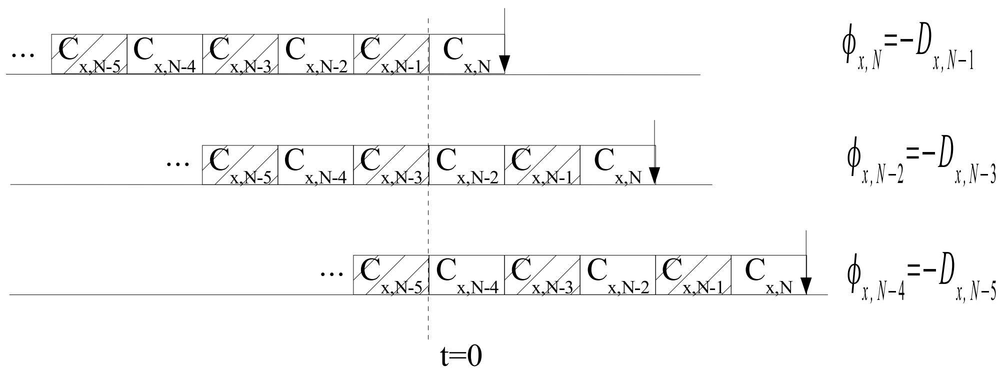


Figure 3.9: Coprocessor Task Phases

the last six elements of its N-element worst-case execution time vector. On the first line, the last coprocessor invocation, $\tau_{x,N-1}$, is being analyzed. The one subtask after the last coprocessor is $\tau_{x,N}$ with phase $\phi_{x,N} = -D_{x,N-1}$. The phase causes $\tau_{x,N}$ to be released at $t = 0$ with no slack. The other subtasks $(\tau_{x,N-2}, \tau_{x,N-4}, \ldots)$ would have positive phase $\phi = T - D_{x,N-1}$. On the second line, the second last coprocessor invocation, $\tau_{x,N-3}$, is being analyzed. Subtasks $\tau_{x,N}$ and $\tau_{x,N-2}$ have phases $\phi_{x,N} = \phi_{x,N-2} = -D_{x,N-3}$. The other subtasks would have positive phase. The analysis is done for each coprocessor invocation, using the appropriate phases.

Algorithm 3.4 describes the feasibility analysis for one coprocessor task with multiple coprocessor blocks. Note that the coprocessor execution times in $\vec{C}$ need not all be the same, accommodating use of multiple coprocessors.

Inside the outer loop of the algorithm is a section labeled "set coprocessor subtask phases". In the first pass ($k = N + 1$), the subtasks of the coprocessor-blocked task are assigned phases of zero. In subsequent passes, k corresponds to the position in the worst-case execution time vector of a coprocessor block. The coprocessor-induced idle that is caused by this coprocessor block is being analyzed. Those subtasks that precede the block

```
Data: task set $\tau$ (coprocessor task $\tau_i$ replaced by $\tau_{i,1}, \tau_{i,3}, \ldots$)
        $N = |\vec{C_i}|$, where $C_i$ is the coprocessor task
Result: schedule feasibility
if $U > 1$ then return "not feasible" ;
/*one pass for phase zero and one pass per block */
for $k$ in $\{N+1, N-1, N-3 \ldots, 2\}$ do
    for $\tau_i \in \tau$ do
        if $|\vec{C_i}| = 1$ then
            $\phi_i = 0$ ;
        else
            /*set coprocessor subtask phases */
            if $k = N+1$ then
                /*first pass - all zero */
                for $a$ in $\{N, N-2, \ldots, 1\}$ do $\phi_{i,a} = 0$ ;
            else
                for $a$ in $\{N, N-2, \ldots, k+1\}$ do $\phi_{i,a} = -D_{i,k}$ ;
                for $a$ in $\{k-1, k-3, \ldots, 1\}$ do $\phi_{i,a} = T_i - D_{i,k}$ ;
            end
        end
    end
    $S' = \cup_{i=1}^{n}\{mT_i + \phi_i + D_i : m = 0, 1, \ldots\} = \{v_1, v_2, \ldots\}$ ;
    $k \leftarrow 1$ ;
    while $v_k < L'$ do
        if $h'(v_k) > v_k$ then return "not feasible" ;
        $k \leftarrow k + 1$ ;
    end
end
return "feasible" ;
```

Algorithm 3.4: Feasibility: One Task, Blocked Multiple Times

Table 3.2: Subtasks of τ_1

Task	Target	Period	Deadline	Worst-Case Exec. Time
$\tau_{1,1}$	processor	$T_{1,1} = T_1 = 30$	$D_{1,1} = D_1 - C_{1,2} - C_{1,3} - C_{1,4} - C_{1,5} = 7$	$C_{1,1} = 3$
$\tau_{1,2}$	coprocessor	$T_{1,2} = T_1 = 30$	$D_{1,2} = D_1 - C_{1,3} - C_{1,4} - C_{1,5} = 9$	$C_{1,2} = 2$
$\tau_{1,3}$	processor	$T_{1,3} = T_1 = 30$	$D_{1,3} = D_1 - C_{1,4} - C_{1,5} = 17$	$C_{1,3} = 8$
$\tau_{1,4}$	coprocessor	$T_{1,4} = T_1 = 30$	$D_{1,4} = D_1 - C_{1,5} = 18$	$C_{1,4} = 1$
$\tau_{1,5}$	processor	$T_{1,5} = T_1 = 30$	$D_{1,5} = D_1 = 20$	$C_{1,5} = 2$

Table 3.3: Phases of τ_1 by Iteration

Iter	k	$\phi_{1,1}$	$\phi_{1,3}$	$\phi_{1,5}$
1	6	0	0	0
2	4	$T_1 - D_{1,4} = 12$	$T_1 - D_{1,4} = 12$	$-D_{1,4} =$ -18
3	2	$T_1 - D_{1,2} = 21$	$-D_{1,2} =$ -9	$-D_{1,2} =$ -9

$(\tau_{i,a} : a = k-1, k-3, \ldots)$ have positive phase. Those that follow $(\tau_{i,a} : a = N, N-2, \ldots)$ have negative phase.

Consider this example: a task set that has one coprocessor task, τ_1, and several other tasks. τ_1 has period $T_1 = 30$, deadline $D_1 = 20$ and worst-case execution time vector $\vec{C}_1 = [3, 2, 8, 1, 2]$. τ_1 blocks on coprocessors twice, with worst-case execution times $C_{1,2} = 2$ and $C_{1,4} = 1$. The subtasks that represent τ_1 are described in Table 3.2. The outer loop of Algorithm 3.4 is repeated three times. In the first iteration, all tasks have phase zero. This is the same processor demand analysis as Algorithm 3.1. The next iteration tests the idle induced by $C_{1,4}$, with $\tau_{1,1}$, $\tau_{1,3}$ and $\tau_{1,5}$ phases as described in Table 3.3. The last iteration tests the idle induced by $C_{1,2}$. If all three iterations pass, then the task set is deemed feasible.

3.2.3 Multiple Tasks Blocking on Coprocessors

Algorithm 3.4 does not extend well to multiple coprocessor-blocked tasks. For example consider two coprocessor-blocked tasks τ_x and τ_y that block on coprocessors once per task instance. If their coprocessor-induced idles overlap, it does not work to assume that both remaining subtasks $\tau_{x,3}$ and $\tau_{y,3}$ have zero slack (as the algorithm does). The matter is further complicated if coprocessors are shared among tasks. This problem of multiple coprocessor-blocked tasks is left as a subject for future research.

3.2.4 Algorithm Benefits and Limitations

In this section, the benefits and limitations of Algorithm 3.3 and Algorithm 3.4 are compared against the simpler analysis of Algorithm 3.2. The comparison is first done for task sets with one coprocessor-blocked task that blocks once per task instance. It is then done for task sets with one coprocessor-blocked task that block multiple times per task instance. These comparisons are followed by a discussion of a simplification made in all the the feasibility analysis algorithms (Algorithms 3.1 – 3.4). A further improvement to Algorithms 3.3 and 3.4 is then suggested.

One Coprocessor-Blocked Task, Blocking Once

This section analyzes the benefit of incorporating one coprocessor invocation into one task, as pertains to feasibility analysis Algorithms 3.2 and 3.3. This is done by first quantifying the change to task worst-case execution time as measured by each algorithm. The resulting benefit to processor utilization is then compared for the algorithms.

- Algorithm 3.2

  This algorithm sums all of the elements of the coprocessor-blocked task's worst-case execution time vector. The difference in task worst-case execution time C_x is the difference between software execution time $C_{x,2}[sw]$ and hardware execution time $C_{x,2}[hw]$:

$$\Delta C_x[3.2] = C_{x,2}[sw] - C_{x,2}[hw].$$

- Algorithm 3.3

  This algorithm does not include the coprocessor execution time in the analysis so the difference in C_x is:

$$\Delta C_x[3.3] = C_{x,2}[sw].$$

The change in processor utilization that results from using a coprocessor in τ_x is equal to the change in execution time over the task period:

$$\Delta U_x = \frac{\Delta C_x}{T_x}.$$

The improvement in processor utilization of Algorithm 3.3 over Algorithm 3.2 is the difference in $\Delta C_x[3.3]$ and $\Delta C_x[3.2]$ over the task period:

$$\begin{aligned}
\Delta U_x[3.3] - \Delta U_x[3.2] &= \frac{\Delta C_x[3.3] - \Delta C_x[3.2]}{T_x} \\
&= \frac{C_{x,2}[sw] - (C_{x,2}[sw] - C_{x,2}[hw])}{T_x} \\
&= \frac{C_{x,2}[hw]}{T_x}
\end{aligned}$$

In Chapter 5, it will be seen that the above analysis is somewhat simplified. When $\tau_{x,2}$ is replaced with a coprocessor invocation, extra time is added to $C_{x,1}$ and $C_{x,3}$ to block and unblock τ_x so that other tasks may execute while the coprocessor is active. This would reduce $\Delta C_x[3.2]$ and $\Delta C_x[3.3]$ equally. The value of $\Delta U_x[3.3] - \Delta U_x[3.2]$ would remain unchanged.

While Algorithm 3.3 takes full advantage of the reduced processor utilization afforded by coprocessor use, it does impose a limitation. In the second iteration, subtask $\tau_{x,3}$ of coprocessor-blocked task τ_x starts at $t = 0$ with no slack. Therefore, all other tasks in the task set must have enough slack to accommodate the execution of $\tau_{x,3}$:

$$\forall |\vec{C}_i| = 1 : D_i \geq C_i + C_{x,3}.$$

This situation may cause problems for tasks with short execution times and tight deadlines. It is suggested that if the limitation above causes a task set to be ruled infeasible by Algorithm 3.3, then Algorithm 3.2 should also be tried since it does not impose this limitation.

One Coprocessor-Blocked Task, Blocking Multiple Times

The observations above are now extended to the coprocessor-blocked task that blocks multiple times.

- Algorithm 3.2

  This algorithm calculates reduced worst-case execution time of the coprocessor task τ_x as:

$$\Delta C_x[3.2] = \sum_{k=2,4,\ldots,N-1} \left(C_{x,k}[sw] - C_{x,k}[hw] \right).$$

- Algorithm 3.4

  This algorithm calculates reduced C_x as:

$$\Delta C_x[3.4] = \sum_{k=2,4,\ldots,N-1} C_{x,k}[sw].$$

The improvement in processor utilization of Algorithm 3.4 over Algorithm 3.2 is the difference in $\Delta C_x[3.4]$ and $\Delta C_x[3.2]$ over the task period:

$$\begin{aligned}
\Delta U_x[3.4] - \Delta U_x[3.2] &= \frac{\Delta C_x[3.4] - \Delta C_x[3.2]}{T_x} \\
&= \frac{\sum_{k=2,4,\ldots,N-1} \left(C_{x,k}[sw] - (C_{x,k}[sw] - C_{x,k}[hw])\right)}{T_x} \\
&= \frac{\sum_{k=2,4,\ldots,N-1} C_{x,k}[hw]}{T_x}
\end{aligned}$$

The limitation noted for the single blocking case is repeated each time τ_x blocks on a coprocessor. The algorithm first checks processor demand with no phases, then with phases corresponding to the ends of coprocessor blocks $\tau_{x,k},\ k = N-1, N-3, \ldots, 2$. For the analysis of each coprocessor block, the following subtask ($\tau_{x,k+1}$) has no slack and must be executed immediately at $t = 0$. So all other regular tasks must have enough slack to accommodate it:

$$\forall |\vec{C}_i| = 1 \text{ and } \forall k \in \{N-1, N-3, \ldots, 2\} : \quad D_i \geq C_i + C_{x,k+1}.$$

Also, if the coprocessor invocation is not the last in the vector, there will be multiple subtasks (i.e. $\tau_{x,k+1}, \tau_{x,k+3}, \ldots$), the first with zero slack and the rest with tight deadlines. For example, in Figure 3.9 when $\phi = -D_{x,N-3}$, $\tau_{x,N-2}$ has zero slack and $\tau_{x,N}$ has slack equal to $C_{x,N-1}$.

Kernel Overhead in Feasibility Analysis

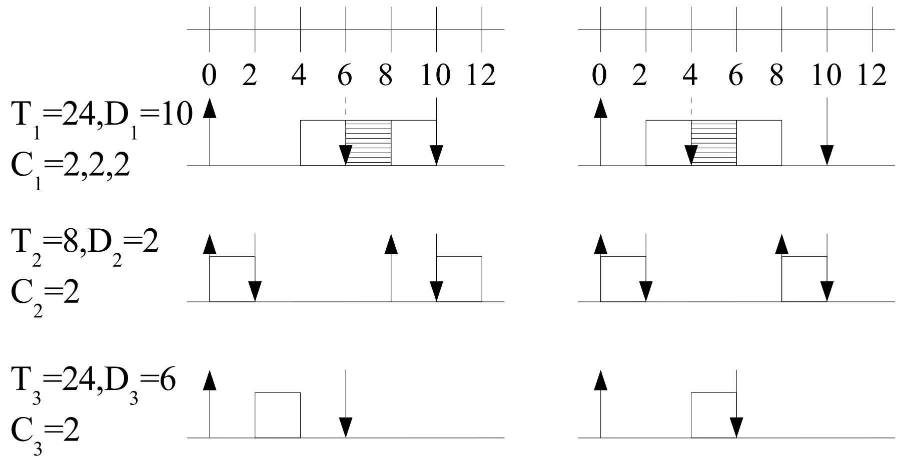
In the feasibility analysis algorithms that have been presented, overhead for scheduling activity has been omitted. This can be interpreted as an ideal scenario in which all scheduling activity takes zero time. This is not, however, reflective of reality. The scheduler is invoked every time that a task is released or terminates. Such events change the set of tasks ready to execute and the scheduler must decide which of the ready tasks to execute. In the case where a task is preempted, its context must be stored, and when a task is re-enabled, its context must be restored.

One method that can be used to incorporate kernel (scheduler) overhead in the feasibility is to add the kernel worst-case execution time twice to the worst-case execution time of each task (for task release and terminate). If the task blocks, for example on a coprocessor or message queue, then the kernel time can be added twice more for each block/unblock pair. This method is used in Chapters 4, 5 and 6 to incorporate kernel overhead into schedule analysis.

Adding kernel overhead in this manner conservatively accounts for the total kernel overhead over an extended period of execution. However, it may under-estimate kernel overhead in certain instances. Consider for example three tasks, τ_1, τ_2 and τ_3 released at times 0, 5 and 10, respectively. If τ_1 has the earliest deadline, it will continue execution until it finishes. However by the time it finishes, the kernel will have been active three times for: release of τ_1, release of τ_2 and release of τ_3. After τ_1 finishes, the kernel is active three more times for: termination of τ_1, termination of τ_2 and termination of τ_3. The total number of kernel invocations is accounted for by adding the kernel overhead twice to each task. However, in the analysis, it only adds two kernel invocations to the worst-case execution time of τ_1, instead of the actual three kernel invocations that occur before τ_1 terminates. This may be an area for further study.

Subtask Deadlines and Feasibility Analysis

The deadline modification method for tasks invoking coprocessors that was presented in Section 3.2.1 is re-evaluated here. A scheduling scenario with one coprocessor-blocked task and two regular tasks is presented in Figure 3.10. In Figure 3.10(a), the deadline of



(a) $D_{1,1} = 6$ (b) $D'_{1,1} = 4$

Figure 3.10: Varying Subtask Deadline

subtask $\tau_{1,1}$ is assigned as described in Section 3.2.1. That is, it is as late as possible. In this example, a deadline is missed when $\tau_{1,3}$ and τ_2 both need to execute at $t = 8$. If the deadline of $\tau_{1,1}$ is decreased by two as shown in Figure 3.10(b), then no deadline is missed.

If the feasibility analysis of Algorithm 3.4 is performed using the maximum $D_{1,1}$, then it would fail in the second iteration when $\phi_{1,3} = -8$. If the feasibility analysis was done with the shorter $D'_{1,1}$ (and shorter $D'_{1,2}$), then the phase of $\tau_{1,3}$ in the second iteration would be $\phi'_{1,3} = -6$. This would give two time units of slack to $\tau_{1,3}$, enabling the task set to pass the feasibility analysis. It is therefore suggested that a heuristic could be used that invokes Algorithm 3.4 repeatedly, while varying the deadline of the coprocessor-blocked subtask $\tau_{x,1}$, to determine whether a $D_{x,1}$ exists that enables the task set to pass the analysis.

3.3 Summary

The feasibility analysis for task sets scheduled by the preemptive EDF policy has been introduced. The algorithm is extended for the analysis of task sets with one task that blocks on coprocessors one or more times. Algorithm 3.4 takes more advantage of the gain in processor utilization than does Algorithm 3.2 since $C_{x,2}$ is not included. However it imposes limitations on the task set that Algorithm 3.2 does not. It is therefore suggested that Algorithm 3.4 and Algorithm 3.2 both be used to verify feasibility of task sets. Algorithm 3.4 is difficult to extend to multiple coprocessor tasks and is left as a subject for further investigation.

It should be noted that it has not been proven that the subtask deadline assignment is an optimal way to schedule the coprocessor-blocked task. Scheduling of the subtasks (and other tasks) is performed by the EDF policy, but the overall scheduling scheme is a modification to EDF and so the optimality claims of EDF cannot be applied without further proof. Also, the proof of Lemma 3.1, proves the worst-case effect of the coprocessor-induced idle but does not prove that this analysis can guarantee feasibility for task sets that include a coprocessor-blocked task.

Chapter 4

Case Study

This chapter describes a case study of a System on Programmable Chip (SoPC). An SoPC is an SoC implemented on a Field Programmable Gate Array (FPGA) rather than an Application Specific Integrated Circuit (ASIC). The implementation of a real-time kernel and application are described. The kernel which schedules by the preemptive Earliest Deadline First (EDF) policy has integrated support for application coprocessors. It is described in Section 4.1. The application simulates and controls an automotive engine running in idle. It consists of several tightly-interacting tasks that are scheduled by the kernel. The application is described in Section 4.2.

When the system is implemented entirely in software, task deadlines are missed and invalid results produced. Two coprocessors are implemented to help meet task deadlines. The first coprocessor implements the cosine function which is used by the application. The cosine coprocessor is described in Section 4.3. The second coprocessor implements EDF scheduling which is used by the kernel. The EDF coprocessor is described in Section 4.4. The effect of the coprocessors on missed deadlines is reported in Section 4.5. The feasibility analysis of Chapter 3 is also applied in this section.

This case study has three benefits:

1. it is a practical example of hardware/software partitioning of application and kernel,
2. the feasibility analysis developed in Chapter 3 can be compared with real results, and
3. case study data is used to test the partitioners in Chapter 7.

4.1 Cs1 Kernel

The SoPC platform is summarized first before describing the kernel that was implemented for this case study. The SoPC is implemented on Altera's Nios Embedded Processor Development Board [4]. The development board has an Altera Apex 20K200E FPGA, 256KB of SRAM, an RS-232 serial connector and a 33 MHz clock chip. The SoPC depicted in Figure 4.1 is implemented on the FPGA. The processor is Altera's Nios [3] soft-core configurable RISC processor. The Nios is configured with a 32-bit data word size and hardware multiply. The UART connects to the RS-232 serial connector for communication with a host development computer. Application downloads and application I/O are done over the RS-232 connection.

Only two kernels were found for the Nios processor. Neither of them were free and neither implemented EDF scheduling, so a custom kernel was implemented in C++ for this case study. The two design goals of the real-time kernel were:

1. Facilitate hardware/software partitioning of applications by providing integrated support for hardware coprocessors.
2. Implement a scheduling policy that achieves high processor utilization. The preemptive fixed-priority policy has sub-optimal processor utilization [67]. Higher processor

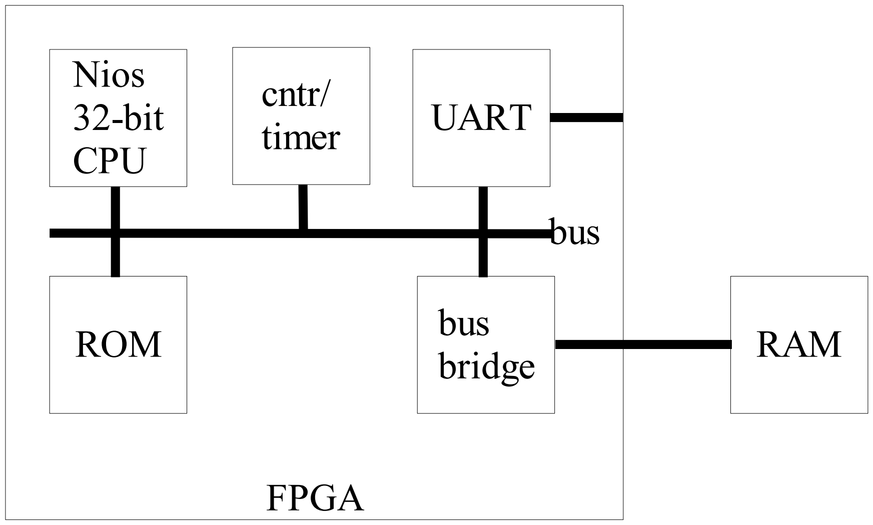


Figure 4.1: SoPC Configuration

utilization can be achieved in theory by using the EDF policy [50].

To keep run-time overhead low, the kernel does not perform dynamic planning: all tasks are created statically at compile time. Any schedule feasibility analysis must be performed off-line.

The uni-processor real-time kernel is called cs1 (for CoScheduler1) [75]. Two types of tasks are supported: periodic and aperiodic. The tasks are scheduled by the preemptive EDF policy: of all active tasks, the task with the earliest deadline is executed first. If another task arrives with an earlier deadline, it preempts the currently executing task. To implement EDF, the cs1 kernel maintains two lists sorted by deadline:

1. the run list - all tasks that are ready to execute, and

2. the timer list - periodic tasks waiting to be released.

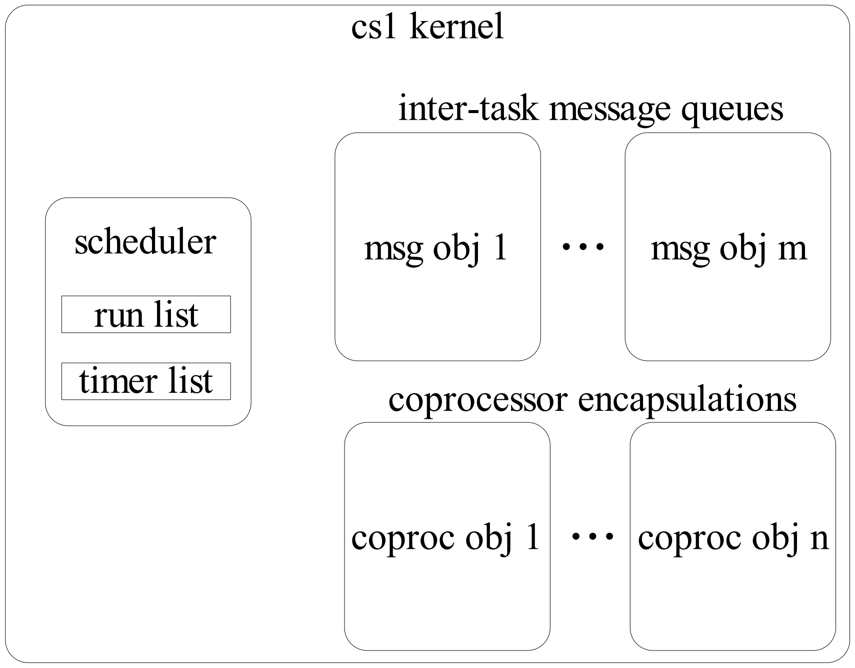


Figure 4.2: cs1 kernel

The run list and timer list are implemented with the min-heap data structure which has $O(\log n)$ insertion and deletion time.

Three system resources are managed by cs1: the system clock, inter-task message queues and hardware coprocessors (Figure 4.2). The system clock is actually a custom 64-bit counter-timer that starts at zero on system initialization. It is programmed by the cs1 kernel to interrupt the processor when the periodic task at the top of the timer list is due for release. For periodic task τ_i, the kernel requires the initial release date (start time) s_i, period T_i and relative deadline D_i. The kernel does not store worst-case execution time for tasks since it is only needed for schedule analysis which is done off-line. Upon release of a periodic task it is placed on the run list and upon termination it is placed on the timer list.

An aperiodic task is released upon the occurrence of an event. The two event types that are supported by the cs1 kernel are:

1. arrival of a message in a message queue, and
2. interrupt by hardware coprocessor.

For aperiodic task τ_i, the kernel requires the identity of the message queue or coprocessor that generates the event and the relative deadline D_i. Upon release of an aperiodic task it is placed on the run list and upon termination it is placed in the wait list of the appropriate message queue or coprocessor (wait lists not shown in Figure 4.2).

A brief explanation of how the kernel integrates coprocessors is given here since it is an uncommon kernel feature. Each coprocessor is represented by a kernel coprocessor object. The object stores information about the coprocessor such as control and status register addresses. Tasks that use the coprocessor must gain access via kernel coprocessor object methods (similar to a critical section). When a task invokes the coprocessor, the task is blocked, freeing the processor for use by other tasks. The coprocessor interrupt is handled by the kernel which checks the coprocessor status, disables the interrupt and unblocks the appropriate task.

4.2 Idle Engine Application

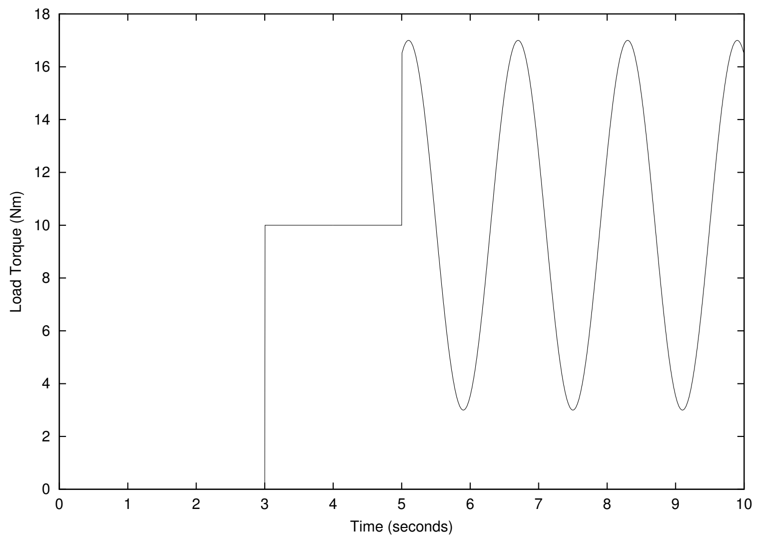
The application selected for this case study consists of a model of an idling automotive engine, environmental input and controllers. The application is not entirely authentic because the environment and the engine model are actually simulated on the same processor as the control application. However it does consist of several tightly interacting tasks, two of which are controllers reacting to events.

The idle engine model, as described in [11] is now summarized. A 4-cylinder automobile engine is modeled in the idle state (out of gear). It is a hybrid model, combining a continuous time system (CTS) with a discrete event system (DES). The components of the CTS are manifold pressure, crankshaft speed and piston position. The cylinders are modeled by an FSM that represents the interleaving between spark ignitions and dead-centers (when a piston reaches the top of its stroke). The DES variables track the torque generated by the pistons. The CTS, FSM and DES are tightly coupled: the FSM transitions in response to CTS events, the DES updates variables based on CTS events and FSM state, and the CTS is affected by DES output.

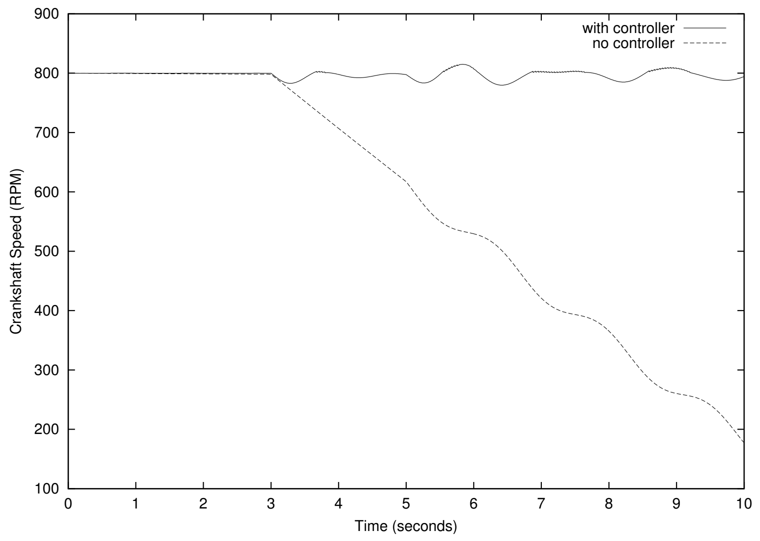
The model has control inputs, throttle angle and spark advance, and environment input, load torque. Both increasing the throttle angle and increasing the spark advance increase the generated torque. The throttle has greater control authority while the spark advance has less authority but has a faster response time. The load torque disturbances for an idle engine are generated by such phenomena as the air conditioning system and the steering wheel servo-mechanism.

The goal is to maintain the crankshaft speed in the range of 800 ± 30 RPM (rotations per minute) under load torque disturbances. The controller devised for this test-case consists of a PID (proportional-integral-derivative) controller for the throttle angle and a P controller for the spark advance. The application consists of four blocks: idle engine model, environment, controller and user interface. The simulated environment input is shown in Figure 4.3(a) and the resulting crankshaft speed is shown in Figure 4.3(b). The crankshaft speed shown is when the controller is used and when it is not used.

The idle engine application was implemented with six tasks configured as shown in Figure 4.4. The directed arrows indicate data dependency. s is start-time, T is period, D is deadline, C is worst-case execution time. All times are given in units of seconds.



(a) Environment Input



(b) Engine Output

Figure 4.3: Model Input/Output

FSM_DES is an aperiodic task that is triggered by dead-center (dc) and spark (spk) events which are generated by the CTS task. FSM_DES is represented as a sporadic task with minimum inter-arrival period T. All other tasks are periodic.

The task worst-case execution times do not include kernel execution which is invoked for every task release and termination. The worst-case execution time of the cs1 kernel, C_{cs1} is 128 μs. When calculating processor utilization U, $2C_{cs1}$ is added to each task C_i. As stated in Chapter 3, a necessary condition for feasibly scheduling a hybrid task set under EDF is $U \leq 1$ [14]. For the idle engine task set, $U = 1.097$ indicating that it is infeasible to schedule this task set by EDF. In practice, it was found that the Throttle task was late 22 times in a 10 second interval.

4.3 Application Partitioning

Two coprocessors were implemented in an attempt to make the idle engine application feasible. The first coprocessor implements a part of the application functionality and is described here. The second coprocessor implements a part of the kernel functionality and is described in Section 4.4.

An examination of the application code reveals that the Environment task invokes the C++ cos() library function each time it executes. The measured worst-case execution time of this function is $1.435ms$. This is a significant amount since it is greater than the worst-case execution time of four of the tasks, and the Environment task is among the most frequently executed tasks. The reason for the high execution time is likely that the software implementation of cosine is based on a polynomial approximation such as a Taylor series. This is the case for the OpenBSD math library implementation [80]. For an approximation of degree 14, at least 6 floating-point multiplications and 5 floating-point additions would

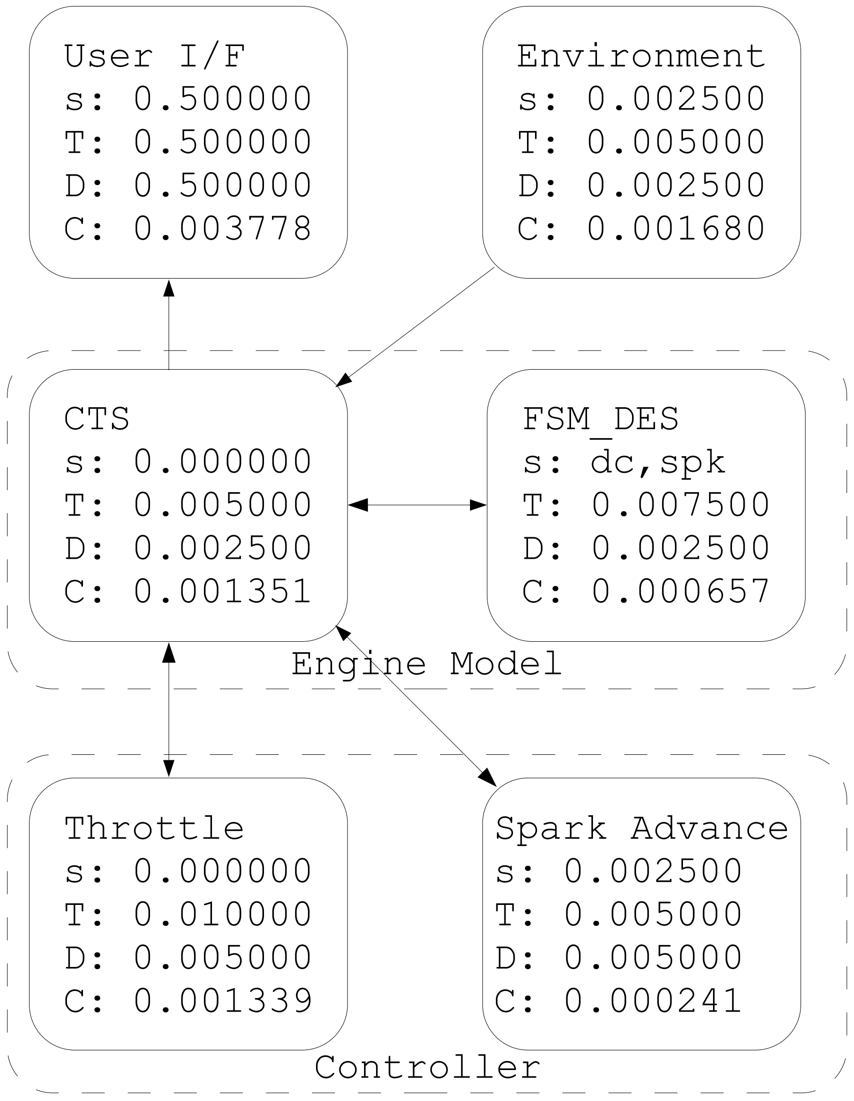


Figure 4.4: Application Configuration

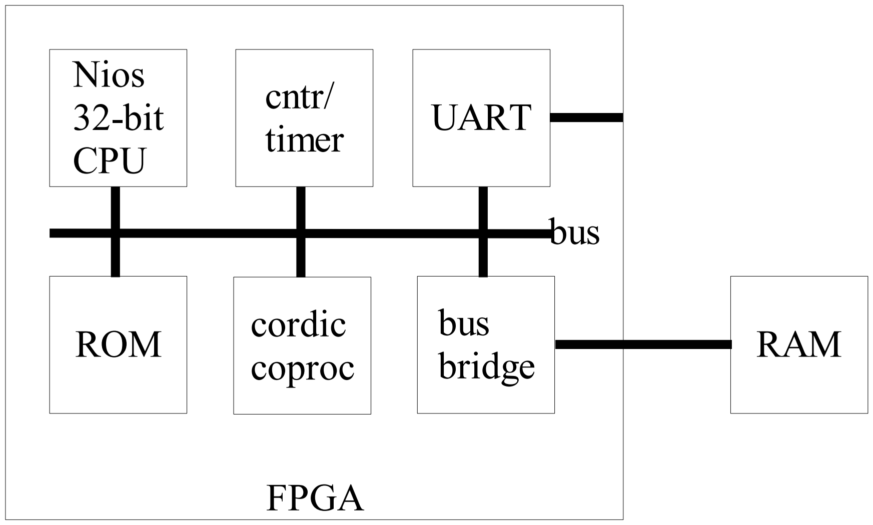


Figure 4.5: SoPC with Cordic Coprocessor

be required. Each floating-point multiplication and addition is time consuming since there is no floating-point hardware and they must be implemented in software.

The coprocessor calculates the cosine function using the Cordic algorithm [102]. This algorithm calculates the cosine of an n-bit fixed point number in n iterations using 3 adders, 2 shifters and a ROM for constants. The coprocessor first converts an IEEE double-precision floating point number to a 64-bit fixed-point number, applies the Cordic algorithm, and then converts the result back to floating point. The coprocessor is implemented with micro-programmed control. The revised SoPC configuration is shown in Figure 4.5.

The implementation results are summarized in Table 4.1. The FPGA has two resource types: logic elements (LE) and embedded system blocks (ESB) providing logic gates and memory [4]. The SoPC with the coprocessor uses approximately 87% of the FPGA's logic resources and 47% of memory resources. The Environment task's worst-case execution time is 4.90 times faster. The effect on missed deadlines and schedule feasibility is discussed in

Table 4.1: Cordic Results

Worst-Case Execution Time	
C++ cos() func	1.435 ms
coproc. cordic cosine	0.081 ms
Env task with cos()	1.680 ms
Env task with cordic	0.343 ms

FPGA Resources		
	LE	ESB
cordic	3840	12
SoPC with cordic	7246	25
FPGA Capacity	8320	52

Section 4.5.

4.4 Kernel Partitioning

The second coprocessor implements a part of the kernel functionality. As described in Chapter 2, the δ Framework facilitates user-directed hardware/software partitioning of the kernel. In another project, the Spring kernel was partitioned by moving some of the scheduler into a coprocessor. Likewise, the goal in partitioning the cs1 kernel is not to transfer the entire kernel into hardware but to choose a strategic part that yields significant system speed-up with small cost. It was decided to implement the EDF scheduling in a kernel coprocessor. The partitioned kernel is called cs2. The cs2 coprocessor replaces management of the run list, the timer list and the system clock (Figure 4.2). The context switching, message passing and coprocessor management remain in the software portion of the kernel. The application programming interface (API) is the same for both kernels.

The pseudocode of the cs1 and cs2 kernels are contrasted in Figures 4.6 and 4.7. The pseudocode describes the activity each time the kernel is invoked. Those parts of cs1 that are omitted from cs2 appear in italics. Those parts of cs1 that are modified in cs2 are underlined.

The design of the cs2 kernel coprocessor is described in more detail than the cordic coprocessor since cordic is a known algorithm [102, 103], whereas the cs2 coprocessor design is original work. The cs2 coprocessor implements the EDF scheduling policy and supports periodic and aperiodic tasks.

To help keep the coprocessor size small, the number of tasks and the task parameters (period, deadline, etc.) are fixed at compile time. To further reduce the coprocessor size, all tasks share the same control logic. For this reason, tasks are processed in a time-slice manner.

In essence, each task in the rotation bids to determine which has the earliest deadline. The first task in the rotation is the Idle task - a pseudo-task employed by the scheduling algorithm. The Idle task writes its deadline ($= 2^{64} - 1$) to the minimum deadline register, d_{min}, and its identifier to a task identity register, tid_{min}. Each successive ready task in the rotation compares its deadline to the value in d_{min}, overwriting d_{min} and tid_{min} with its own values if its deadline is earlier. The last task in the rotation is another pseudo-task called the Irq task. At the end of every rotation the Irq task compares tid_{min} with the value from the previous rotation. If the task with the earliest deadline has changed, the processor is interrupted to indicate that a context switch is required. When no application tasks are ready for execution, the Idle task wins the bidding and is scheduled. Otherwise the task with the earliest deadline wins.

The cs2 coprocessor is mapped into the processor's address space and communicates

```
1.track task & kernel execution times
2.check current task status for:
  a) terminate
     · remove from run list
     · if task ready for release
        · reset and insert in run list
     · else
        · insert in appropriate list (timer, msg reader,
          coproc service)
  b) task blocked
     · remove from run list & append to appropriate list
       ( msg writer, msg reader,coproc setup, coproc
         service)
  c) unblocked other task
     · dequeue other task from appropriate list(msg
        writer, msg reader, coproc setup, coproc service)
     · insert in run list
3.check timer list
  · remove ready tasks from timer list
  · reset popped tasks and insert in run list
4.check coprocessors
  · if irq asserted
     · mask irq
     · if service list not empty
        · dequeue task, reset (if needed), insert in run list
     · else
        · set coprocessor ready for service
5.set up timer for next periodic task release
6.read next task from head of run list
7.perform context switch (if needed)
```

Figure 4.6: cs1 Kernel Pseudocode

```
1.track task & kernel execution times
2.if current task terminated
  · communicate event to cs2 coproc
  · if aperiodic task ready for release
    · reset and communicate event to cs2 coproc
  · else
    · append to appropriate list
      (msg reader, coproc service)
3.check coprocessors
  · if irq asserted
    · mask irq
    · if service list not empty
      · dequeue task, reset (if needed)
      · communicate event to cs2 coproc
    · else
      · set coprocessor ready for service
4.read next task from cs2 coproc
5.perform context switch (if needed)
```

Figure 4.7: cs2 Kernel Pseudocode

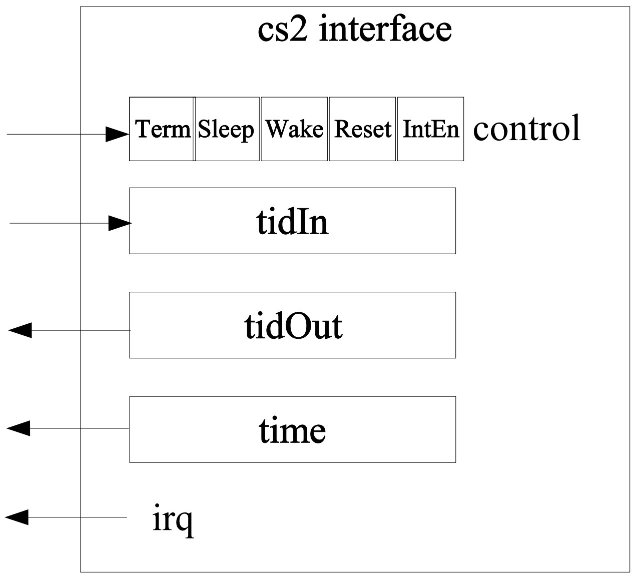


Figure 4.8: cs2 coprocessor interface

over the system bus. The cs2 coprocessor interface is depicted in Figure 4.8. To communicate to the cs2 kernel that a context switch is required, the cs2 coprocessor asserts the interrupt request (*irq*) and writes the task identifier of the next task to run in the *tidOut* register. The cs2 kernel communicates changes in task status to the cs2 coprocessor by writing the task identifier to the *tidIn* register and writing the appropriate code to the control register (terminate, sleep, wake).

The structure of the cs2 coprocessor is shown in Figure 4.9. The task set always consists of τ_0, the Idle task, and τ_{n+1}, the Irq task. τ_1 to τ_n are the application tasks. For demonstration purposes, a periodic task (τ_1) and an aperiodic task (τ_2) are shown in the figure. The periodic task has 4 constants: task type (periodic), start time (s), period (T), and relative deadline (D). It also has 3 variables: state, release date (r), and absolute deadline (d). The other task types (aperiodic, idle, irq) require a subset of this information.

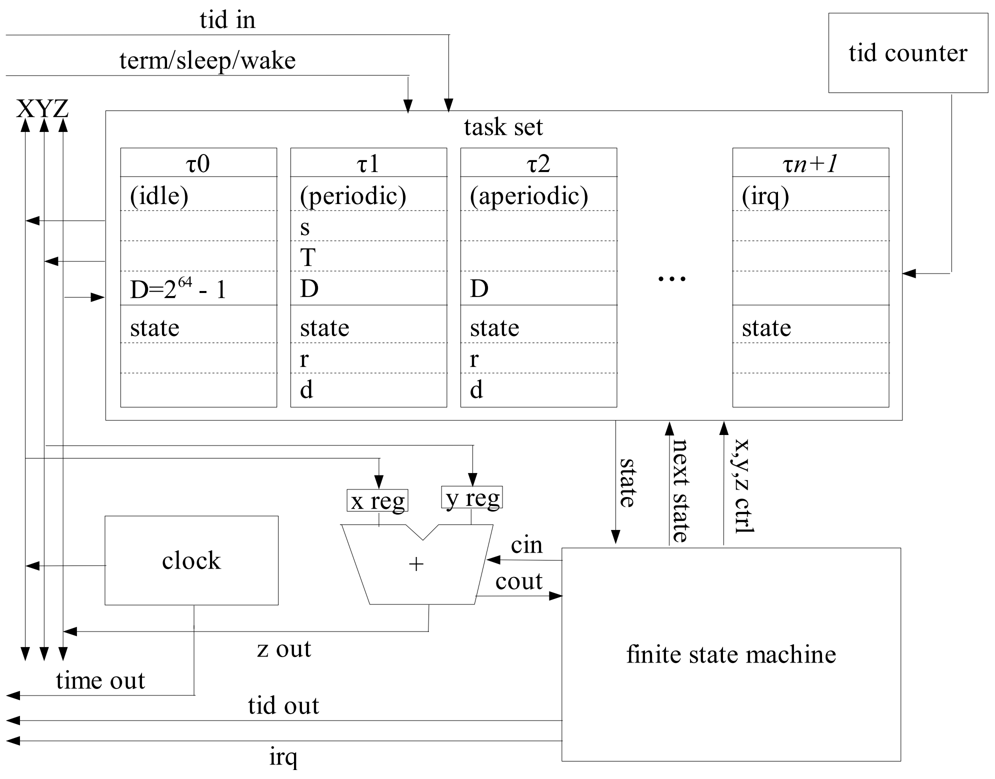
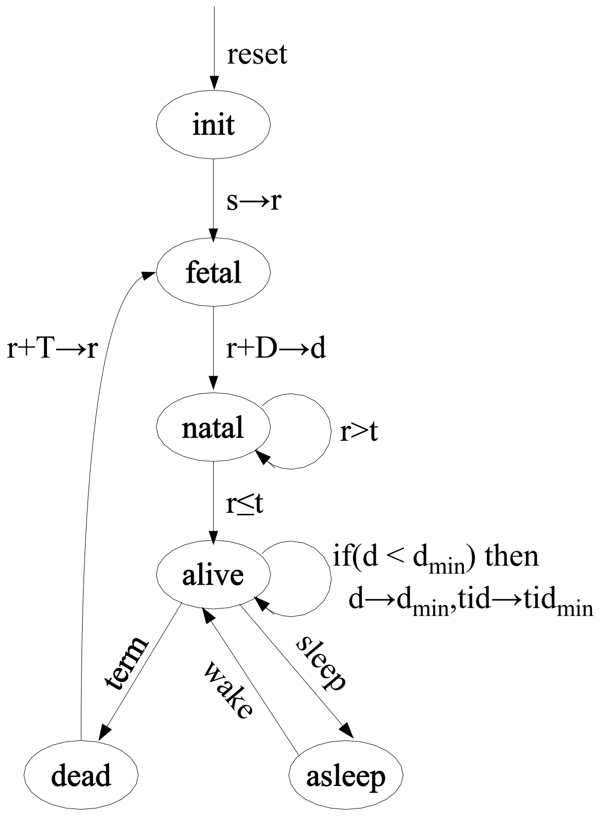


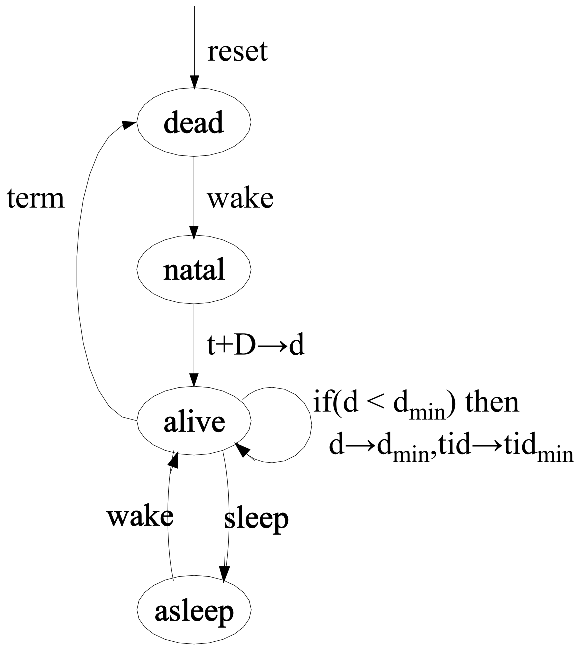
Figure 4.9: cs2 coprocessor structure

Each task is governed by a finite state machine (FSM). The FSMs of the periodic, aperiodic, Idle and Irq task types are shown in Figure 4.10. For example, each of the periodic tasks would be governed by the FSM in Figure 4.10(a). This doesn't mean however that all of the periodic tasks transition through the same states simultaneously. They each have unique state but share the FSM control and data-path logic by being processed in time-slice manner. This reduces the size of the cs2 coprocessor.

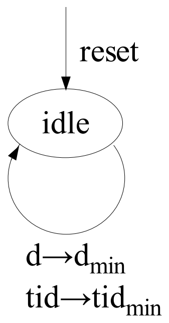
The size of the cs2 coprocessor depends on the number and type of tasks. The size and performance of the cs2 coprocessor is analyzed in detail in [76]. In Table 4.2, results



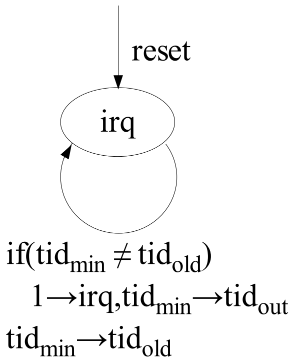
(a) Periodic



(b) Aperiodic



(c) Idle



(d) Irq

Figure 4.10: Task Type FSMs

Table 4.2: Cs2 Results

Worst-Case Execution Time	
cs1	128 μs
cs2	66 μs
cs2 coproc	4.56 μs

FPGA Resources		
	LE	ESB
cs2 coproc	2496	1
SoPC with coproc	5242	14
FPGA Capacity	8320	52

are reported for the cs2 kernel and coprocessor configured for the idle engine application. Impact on schedule feasibility is analyzed in Section 4.5.

4.5 Feasibility Results

The effect that coprocessors have on the idle engine application's feasibility is evaluated for four cases: 1) SoPC with no coprocessor, 2) SoPC with cordic coprocessor, 3) SoPC with cs2 coprocessor, and 4) SoPC with cordic and cs2 coprocessors. The feasibility analysis of Algorithm 3.4 is used to test for feasibility. In the two cases where the cordic coprocessor is used, Algorithm 3.2 is also used. (Both algorithms work the same when no application coprocessor is used as in the first and third cases.)

SoPC - no coprocessor

The kernel worst-case execution time, $C_{cs1} = 128\mu s$, is added twice to each task's worst-case execution time: $C_i = C_i + 2C_{cs1}$. As a result, $U = 1.097$. The task set is deemed infeasible by the analysis of Algorithm 3.4. When the application was run for 10 seconds on the SoPC, the Throttle task was late 22 out of 1000 times, corroborating the feasibility analysis. (During a 10 second run of the application, over 8000 tasks instances are executed. The absence of missed task deadlines is not a proof of feasibility but the number of missed deadlines is expected to be roughly indicative of the difficulty of scheduling the task set.)

SoPC - cordic coprocessor

The Environment task worst-case execution time vector is:

$$\vec{C}_{env} = \langle 0.000074, 0.000081, 0.000269 \rangle \, .$$

The task first executes on the processor for 74 μs, then executes on the coprocessor for 81 μ and then executes on the processor again for 269 μs. For the feasibility analysis, the Environment is represented by subtasks $\tau_{env,1}$ and $\tau_{env,3}$ which execute on the processor and $\tau_{env,2}$ when executes on the coprocessor. The kernel worst-case execution time C_{cs1} is added twice to each software task's worst-case execution time. The Environment subtasks have parameters:

	T_i	D_i	C_i
$\tau_{env,1}$	0.005000	0.004394	0.000330
$\tau_{env,2}$	0.005000	0.004475	0.000081
$\tau_{env,3}$	0.005000	0.005000	0.000525

Following Algorithm 3.4:

- $U = 0.881101$
- Processor demand analysis ($\forall \tau_i : \phi_i = 0$):

- $L = 0.039564$
- $S = \{0.002500, 0.004394, 0.005000, \ldots, 0.039394\}$ ($|S| = 25$)
- $h(0.002500) = 0.002520 \quad \longrightarrow$ infeasible

Since processor utilization is less than one, processor demand can be analyzed. Analysis is first done with all task phases equal zero. The synchronous busy period has length $L = 0.039564$. The set of deadline events in $[0, L)$ is constructed. Processor demand analysis of the first event in S, $v = 0.0025$, shows that $h(v) > v$ and the task set is deemed infeasible.

When Algorithm 3.2 is used, the processor utilization is higher: $U = 0.897301$. This is because the coprocessor time is included in the task worst-case execution time. It fails at the same point that Algorithm 3.4 fails.

When the application was run on the SoPC for 10 seconds, no task deadlines were missed. This apparent discrepancy can be resolved by observing that the analysis is conservative: it ignores task start times and also assumes that each task has execution time equal to its worst-case for every invocation. While the analysis may return "infeasible" for a feasible task set, it should never return "feasible" for an infeasible task set.

SoPC - cs2 coprocessor

The kernel worst-case execution time, $C_{cs2} = 0.000066s$ is added twice to each task's worst-case execution time: $C_i = C_i + 2C_{cs2}$.

Following Algorithm 3.4 (or Algorithm 3.2):

- $U = 0.993720$
- Processor demand analysis ($\forall \tau_i : \phi_i = 0$):

- $L = 0.299680$
- $S = \{0.002500, 0.005000, 0.007500, \ldots, 0.297500\}$ ($|S| = 178$)
- $h(0.002500) = 0.002272$

  $h(0.005000) = 0.005928 \quad \longrightarrow$ infeasible

U is higher than for the cordic coprocessor case above and L is much longer, resulting in a larger event set to check. At the first event, $v = 0.0025$, $h(v) \leq v$ meaning that all tasks with deadlines by 0.0025 have time to execute. However at the second event, $v = 0.0050$, processor demand exceeds available time and the task set is deemed infeasible.

When the application was run on the SoPC for 10 seconds, the Throttle task was late 17 times out of 1000. This result supports a view that the cordic coprocessor contributed more to task feasibility than the cs2 coprocessor. Examining the analytical results, U for the cs2 coprocessor case is very close to one, which may indicate that the task set is difficult to schedule.

SoPC - cordic and cs2 coprocessor

The cordic and cs2 coprocessor did not fit onto the FPGA used for this case study. However the feasibility analysis can still be conducted. The kernel worst-case execution time is $C_{cs2} = 0.000066s$. This changes the Environment subtask parameters since C_{cs2} is added twice to each of $\tau_{env,1}$ and $\tau_{env,3}$. The resulting parameters are:

	T_i	D_i	C_i
$\tau_{env,1}$	0.005000	0.004518	0.000206
$\tau_{env,2}$	0.005000	0.004599	0.000081
$\tau_{env,3}$	0.005000	0.005000	0.000401

Following Algorithm 3.4:

- $U = 0.752720$
- Processor demand analysis ($\forall \tau_i : \phi_i = 0$):
  - $L = 0.019071$
  - $S = \{0.002500, 0.004518, 0.005000, \ldots, 0.017500\}$ ($|S| = 10$)
  - for each $v \in S$, $h(v) \leq v$:

    $h(0.002500) = 0.002272$

    $h(0.004518) = 0.002478$

    $h(0.005000) = 0.004723$

    $\ldots$

    $h(0.017500) = 0.014181$
- Processor demand analysis ($\phi_{env,1} = T - D_{env,2}$ $\phi_{env,3} = -D_{env,2}$):
  - $L = 0.019472$
  - $S = \{0.000401, 0.002500, 0.004919, \ldots, 0.017500\}$ ($|S| = 14$)
  - $h(0.000401) = 0.000401$

    $h(0.002500) = 0.002673 \quad \longrightarrow$ infeasible

The task set passes analysis with all task phases equal zero. For the second iteration of Algorithm 3.4, $\tau_{env,1}$ and $\tau_{env,3}$ have phases as indicated above. The first event, $v = 0.000401$ is the deadline of $\tau_{env,3}$ which has zero slack. At the second event, $v = 0.0025$, $h(v) > v$ and the task set is deemed infeasible. (The worst-case execution time of the kernel was experimented with and the analysis passed when the kernel time was reduced to $C_{cs2} = 37\mu s$ - approximately half of the true value.)

Following Algorithm 3.2:

- $U = 0.768920$
- Processor demand analysis ($\forall \tau_i : \phi_i = 0$):
  - $L = 0.019395$
  - $S = \{0.002500, 0.005000, 0.007500, \ldots, 0.017500\}$ ($|S| = 7$)
  - for each $v \in S$, $h(v) \leq v$:
    $h(0.002500) = 0.002272$
    $h(0.005000) = 0.004804$
    $h(0.007500) = 0.006287$
    $\ldots$
    $h(0.017500) = 0.014424$

Again processor utilization is higher, as expected. Although its synchronous busy period is longer ($L = 0.019395$), it tests fewer deadline events than the first iteration of Algorithm 3.4 because the Environment task is treated as one task, rather than two subtasks. Even though the processor demand at most events is higher than in Algorithm 3.4, all deadline events pass the analysis, indicating that the task set is feasible. Note that failing the analysis of Algorithm 3.4, does not mean that the task set is infeasible; only that the analysis cannot guarantee that there will be no missed deadlines.

The four test cases that were evaluated show that, as expected, the feasibility analysis is conservative, rejecting tasks sets as infeasible that may in fact be feasible (i.e. did not miss deadlines during testing). Furthermore, in the last case, Algorithm 3.2 passed while Algorithm 3.4 failed, showing the importance of using both algorithms. Note also that as U decreased, the number of missed deadlines also decreased (see Table 4.3).

Table 4.3: U vs Missed Deadlines

Case	Processor Utilization		Missed Deadlines
	Algorithm 3.4	Algorithm 3.2	
SoPC - no coproc.	$U = 1.097$	$U = 1.097$	22
SoPC - cs2	$U = 0.993720$	$U = 0.993720$	17
SoPC - cordic	$U = 0.881101$	$U = 0.897301$	0
SoPC - cordic + cs2	$U = 0.752720$	$U = 0.768920$	n/a

Table 4.4: Coprocessor Comparison

	Δt	ΔU	LE	ESB	$(\Delta U/\text{LE}) \times 10^6$
cordic	$1.081ms$	0.2162	3840	12	56.30
cs2	$62\mu s$	0.1036	1836	1	56.42

4.5.1 Coprocessor Evaluation

Two methods were used above to evaluate the effect a coprocessor had on schedule feasibility: implementation testing and feasibility analysis. The hardware/software partitioners that are developed in the next two chapters must evaluate many coprocessors in varying combinations. For large problems, the implementation and testing of every coprocessor in every combination would be too time consuming. The feasibility analysis should not require as much effort but is also expected to be too time consuming. Therefore a faster indicator of task feasibility is needed. From the results above, it was noted that as processor utilization decreased, the number of missed deadlines also decreased. In Table 4.4, the cordic and cs2 coprocessor are compared by ΔU.

Δt for cordic is calculated from the difference in worst-case execution times for the Environment task from Table 4.1 and by subtracting $2C_{cs1}$ since two extra kernel invocation are required to block and unblock the task. ΔU is calculated by dividing Δt by the Environment task period, $T_i = 5ms$.

Δt for cs2 is calculated from the difference in worst-case execution times for the kernels from Table 4.2. The size of the coprocessor is adjusted by the size of the counter-timer since the cs2 coprocessor eliminates the need for it. The counter-timer coprocessor uses 660 LEs and 0 ESBs. ΔU for the cs2 coprocessor is calculated as:

$$\Delta U = \sum_{\tau_i \in \tau} \frac{2\Delta t}{T_i}.$$

The kernel is invoked twice for each task invocation for release and termination, so Δt cs2 is saved twice for each task invocation. $2\Delta t$ cs2 is divided by each task's period and summed to calculate ΔU.

ΔU for cordic is approximately twice ΔU for cs2. This agrees with the experimental results where the SoPC with cordic coprocessor had no late tasks, while the SoPC with the cs2 coprocessor reduced late tasks from 22 to 17. However the cordic coprocessor requires approximately twice as many FPGA resources as the cs2 coprocessor. In a system where multiple coprocessors may be used, it might be useful to scale ΔU by the coprocessor size to estimate gain per LE. The cordic coprocessor uses 46% of the FPGA LEs and 23% of ESBs. The cs2 coprocessor uses 22% of the FPGA LEs and 2% of ESBs. In both cases a larger percentage of LEs than ESBs are used to implement the coprocessor. Therefore ΔU is scaled by LEs. It is also multiplied by 10^6 for easier reading. The resulting values are very close: 56.3 for cordic and 56.4 for cs2. Both ΔU and ΔU/LE are considered as gain measures for the partitioning heuristic described in Chapter 6.

It is interesting to note that ΔU calculations for both coprocessors depend not only on the task/kernel that uses the coprocessor but also on other parameters. For the cordic coprocessor, Δt depends on the kernel worst-case execution time as well as the change to the Environment task. For the cs2 coprocessor, ΔU depends on the change to kernel worst-case execution time and the number of kernel invocations which is determined by the task set. This dependence of ΔU on both the task set and kernel is incorporated into the partitioning heuristic in Chapter 6.

4.6 Summary

A case study has been performed on an SoPC with a real-time kernel and application. An all-software implementation failed the EDF feasibility analysis and in testing, the Throttle task was late 22 times out of 1000. Hardware/software partitioning of the application and the kernel was demonstrated. The larger application coprocessor was successful in eliminating missed deadlines. However neither coprocessor caused it to pass the feasibility analysis which is conservative. However it was noted that as processor utilization decreased, the number of missed deadlines also decreased. Two metrics for fast comparison of coprocessors have been suggested: U and U/LE. Both metrics facilitate the comparison of coprocessors, whether kernel coprocessors or application coprocessors. These metrics are considered in development of the hardware/software partitioning heuristic in Chapter 6.

Chapter 5

Hardware/Software Partitioning

This chapter defines the hardware/software partitioning problem being studied in this dissertation. In Section 5.1, the problem is defined with its accompanying assumptions and limitations. In Section 5.2, a non-linear programming (NLP) model is developed. A heuristic solution to the problem is proposed in Chapter 6. Results from evaluating the NLP model and the heuristic solver are presented in Chapter 7.

5.1 Problem Definition

The hardware/software partitioning problem identified in this chapter has two key features which are listed here.

1. *Partition the combined application and kernel.*
   Several codesign systems were reviewed in Chapter 2 that performed hardware/software partitioning of the application. Other work was reviewed that explored hardware/software implementation of the kernel (or operating system). However the two activities were not combined.

2. *Partition concurrent systems so as to produce feasible schedules.*

   Of the systems in Chapter 2 that combined scheduling with partitioning, none examined concurrent applications. Any systems that managed concurrent applications, synthesized schedulers after the partitioning stage. They did not integrate scheduling of concurrent applications with partitioning.

The hardware/software partitioning problem discussion is divided into four parts. In Section 5.1.1, the input format of the application and kernel is discussed. The implementation target input requirements are presented in Section 5.1.2. Assumptions made in modeling edge and node costs are discussed in Section 5.1.3, followed by a declaration of the problem in Section 5.1.4.

5.1.1 Problem Input: Application and Kernel

The focus of this thesis is on partitioning and scheduling. For this reason, the internal representation (IR) is described in detail but the method of system specification is left open. Any specification method that contains the information needed to build the IR should be compatible with this work. The System Level Intermediate Format (Slif) was chosen as the IR for the following four reasons.

1. Node granularity - The nodes of a Slif graph represent application behaviour at the function level. This is fine enough that a task can be partitioned into multiple nodes (see case study Slif graphs in Appendix B.2). It is coarse enough to allow simple calculation of task execution times: edge counts and node execution times can be easily combined to calculate task execution time. If a finer granularity was used, such as the basic block or operation level, then conditional execution of loops and choice statements would complicate calculation of execution time. It is also a convenient

level to represent library functions for which no source code is available (further decomposition would be difficult without source code). However this representation can limit the partitioner's options as becomes apparent when transforming the case study data for partitioning in results Section 7.1.

2. Node and edge annotations - The annotations incorporate data needed by the partitioner such as node execution times, node sizes, and edge counts.

3. Benchmarks - Six benchmarks are available that are based on real applications. Unfortunately they are all uni-process and do not extend easily to concurrent problem representation.

4. Generic problem generator - A public domain utility, called gpslifgen, is available that generates generic problem instances based on statics obtained from the benchmarks.

The Slif annotations used in the model in Section 5.2 are: node worst-case execution times (hardware and software), node size (hardware and software) and edge count. Edge count represents the number of times, per source node execution, that the edge's destination node is invoked.

There is one Slif graph for the kernel and one Slif graph for each application task. Application task Slif graphs can have nodes in common: for example, two tasks may invoke the same subroutine. It is expected that the node annotations will be identical if the node appears in multiple Slif graphs. Kernel nodes cannot be shared with application tasks. This is an important simplification in developing the heuristic in Chapter 6. It would also probably occur without such a restriction because kernel nodes implement scheduling policy and other kernel services which are not likely to be implemented in application task code.

The Earliest Deadline First (EDF) policy is used for reasons described in Chapter 3, so task scheduling parameters are also needed in the problem input. For each task τ_i, relative deadline D_i and period (or minimum inter-arrival rate for sporadic tasks) T_i must be specified. Worst case execution time C_i for task τ_i is not specified: it is calculated from the Slif graph and depends on how the nodes are partitioned. Explicit scheduling information is not required for the kernel: the same kernel is invoked for any scheduling activity such as task release, block, unblock and termination.

5.1.2 Problem Input: Implementation Target

In order to partition the application and kernel, the implementation target onto which they are being mapped needs definition. The following definition of the target technology is assumed:

1. there is one processor with fixed throughput,
2. coprocessors are memory mapped (see discussion in Section 5.1.3),
3. programmable logic for coprocessors has limited capacity, and
4. memory for program and data may be limited.

These assumptions of the target technology are applicable to System on Chip (SoC) and System on Programmable Chip (SoPC) systems. In Chapter 4, a case study of an SoPC is described. A single processor connects to a system bus to which memory, peripherals and coprocessors connect. Results from the case study indicate that a small part of the application and kernel functionality can fill the programmable logic to capacity. Whether memory is a limiting factor depends on the configuration of the SoC/SoPC. If the memory is embedded memory, derived from the FPGA resources, it can be a limiting factor. If the memory is external, as in the SRAM on the development board of the case study, it may not be a limiting factor. Another way that memory can be expanded in an SoC is to integrate logic chips and memory chips in one package [104].

Multi-processor systems are not included in this definition of the hardware/software partitioning problem but are discussed as a possible extension of the work in Chapter 8.

The implementation parameters required by the partitioner are: processor throughput (clock speed or retired instruction rate), logic size (logic elements or gate equivalents) and memory size (embedded memory blocks or bytes of external memory). In addition, Slif nodes must be measured for execution time and size on the target implementation - both software and hardware. Other resources could be measured and constrained as discussed in Section 5.2.5.

5.1.3 Problem Modeling: Assumptions

A key assumption of the model developed for the hardware/software partitioning problem is that uncut edges do not incur additional cost but that cut edges do. This assumption is consistent with the other hardware/software partitioning tools surveyed in Chapter 2. To justify this assumption, four cases are considered. Before examining the cases recall that in a Slif graph an edge represents an invocation of one function by another. (It may also represent an access to a global variable but that interpretation is not used in this model.) With two nodes per edge and two implementation targets there are four permutations to consider. These four cases are discussed as they apply to task edges. Kernel edges are discussed later in this section.

1. **software to software** (uncut)

   Parameters are pushed onto the stack by the invoking function which then executes a “branch to subroutine” instruction. The invoked function retrieves parameters from the stack and when finished, optionally puts a result on the stack and executes a "return from subroutine" instruction. These edge costs are included in the invoking and invoked node costs.

2. **hardware to hardware** (uncut)

   The invoking node possibly latches output parameters and asserts a control signal to enable the invoked node. The invoked node reads the output signals from the invoking node, generates an output and asserts another signal to indicate that the result is valid. These edge costs are included in the invoking and invoked node costs.

3. **software to hardware** (cut)

   Instead of pushing parameters onto the stack, the invoking function writes them to memory-mapped registers on the coprocessor. The coprocessor reads input parameters from the registers and when finished, optionally puts a result in a memory-mapped register and interrupts the processor. The invoking function reads the result from the register. These edge costs are included in the source and destination node costs. Furthermore, the cost for the invoking node should be similar to the software-software case and the cost for the invoked node should be similar to the hardware-hardware case.

   There are however other costs. The interaction of software with coprocessor is modeled on the case study kernel. Tasks block on coprocessor invocation and the kernel handles the coprocessor interrupt, unblocking the invoking task. Therefore a software node invoking a hardware node (a cut edge) incurs the additional cost of two kernel invocations.

4. **hardware to software** (cut)

   Instead of latching parameters into an internal register, the coprocessor latches them

Table 5.1: Edge Cost Summary

Slif Graph	Edge Direction	Cut?	Included Cost: In Src. Node	Included Cost: In Dst. Node	Extra Cost
Task	sw → sw	no	param. push, call, result pop	param. pop, result push, return	
	hw → hw	no	”	”	
	sw → hw	yes	”	”	block unblock
	hw → sw	yes	”	”	block unblock
Kernel	any	n/a	”	”	

into a memory-mapped register and asserts the interrupt request. As in the case study, the kernel handles the interrupt and unblocks or releases the appropriate task. That task reads the input parameters from the memory-mapped registers and when finished writes the result to a memory-mapped register and writes a value to the coprocessor control register to transfer control. The task would then block or terminate. In this case the cost to the invoking node is similar to the hardware-hardware case and the cost to the invoked node is similar to the software-software case. Again the cut edge incurs the additional cost of two kernel invocations.

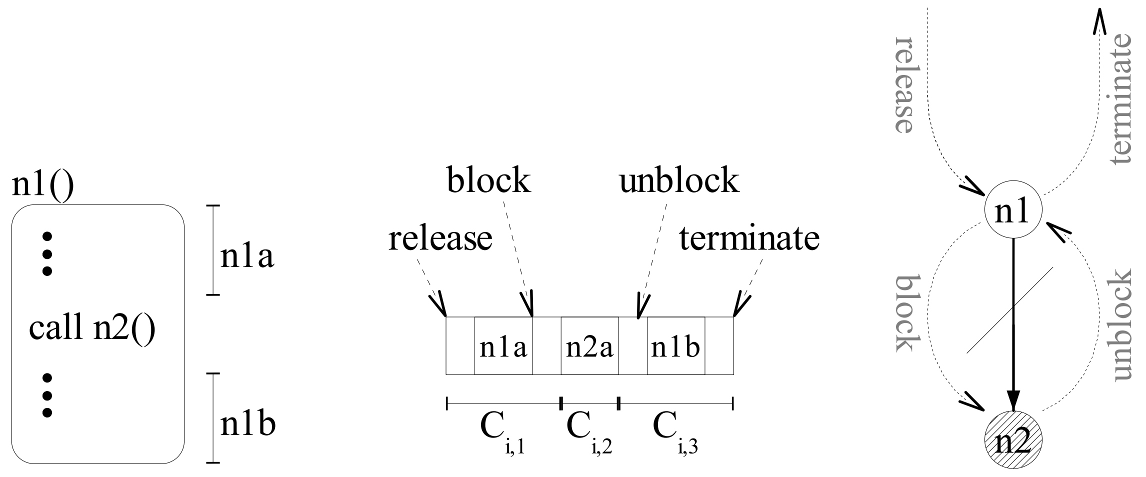
Based on these four cases, it is assumed that the cost of uncut edges is incorporated into the invoking and invoked node costs, and that cut edges incur the additional cost of two kernel invocations. This discussion is summarized by Table 5.1.

The discussion of edge costs has so far only dealt with application task edges. Kernel edges are assumed to have no cost, whether they are cut or uncut. This is because the kernel does not block on a coprocessor invocation. Instead it busy-waits on the coprocessor until the coprocessor finishes. This busy-waiting also eliminates the overhead of an extra interrupt-service call to handle the coprocessor interrupt.

An example is introduced in Figure 5.1 that shows the relationship between source code and partitioned Slif graphs. It also shows how Slif graphs relate to the time vectors $\vec{C_i}$ used in Chapter 3. Figure 5.1(a) shows a source code representation for a function, n1(), that invokes another function, n2(). The part of n1() that executes before invoking n2() is labeled n1a and the part that executes after is labeled n1b.

Figure 5.1(b) shows how this task would be represented as a time vector $\vec{C_i}$ if n2() was implemented on a coprocessor and n1() remained in software. Time is added before and after n1a for task release and task block. The combined times of n1a, release and block is the first element, $C_{i,1}$, in the vector. The execution time of n2 on the coprocessor is the second element, $C_{i,2}$. The combined times of n1b, unblock and terminate is $C_{i,3}$.

In Figure 5.1(c) this task is represented by a Slif graph for hardware/software partitioning. The Slif edge direction indicates which node invokes the other. However, each edge is actually traversed twice: once to transfer control to the invoked node and once to transfer control back to the invoking node. Dashed lines have been added to the Slif graph to indicate when an edge causes kernel activity. Unshaded nodes are assigned to software and shaded nodes to hardware. Cut edges are indicated by a slash through the edge. Node n1 is a indivisible unit with execution time equal to the sum of n1a and n1b. Since node n1 is the root of the Slif graph and is implemented in software (unshaded), kernel overhead is incurred once to release the task and once to terminate the task. Since node n2 is implemented in hardware (shaded), kernel overhead is incurred once to block n1 and once to unblock n1. Note that the Slif graph specification does not include enough information to generate the worst-case execution time vector for EDF feasibility analysis:



(a) Source Code View (b) Time Vector View (c) Slif Graph View

Figure 5.1: Partitioned Task with One Cut Edge

n1 is annotated with only the sum of n1a and n1b, so it cannot be determined at what point in execution the outgoing edge is traversed.

Figure 5.2 shows another partitioned Slif graph. In this case, the root node is bound to hardware. For the partitioning model, it is assumed that a root node implemented in hardware would be released independent of the kernel and so no release/terminate pair are needed to enter/exit the root node. The two children of the root are bound to software and are shown being released/terminated for each node. Depending on how the graph is transformed into source code, the software nodes could be combined into one task or divided into multiple tasks. In either case, each traversal of a cut edge requires two kernel invocations. Two uncut edges (software to software, and hardware to hardware) are also shown: these do not require kernel activity.

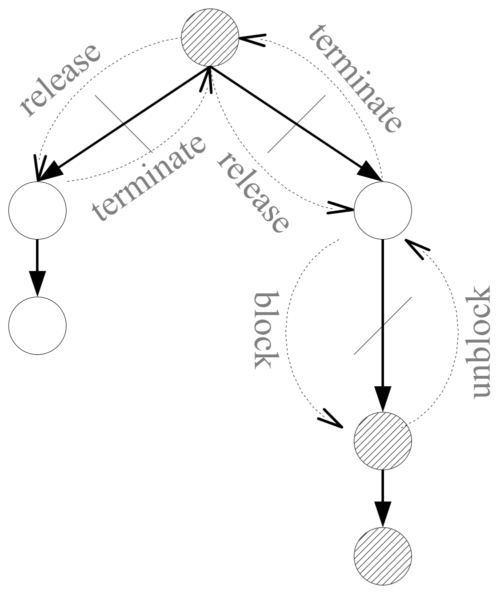


Figure 5.2: Partitioned Task with Multiple Cut Edges

5.1.4 Problem Declaration

The hardware/software partitioning problem is declared as a constraint satisfaction problem:

Declaration 5.1. *Given one processor with fixed throughput, limited programmable logic, and possibly limited program/data memory, partition the application and kernel such that all deadlines are met when scheduled by the EDF policy.*

This definition of the hardware/software partitioning problem and the assumptions and limitations above are encapsulated by the non-linear programing (NLP) model that follows.

5.2 Non-Linear Programming Model

The NLP model of the hardware/software partitioning problem is presented in four parts: input parameters, variables, objective function and constraints. The goal is to partition the system so that it can be feasibly scheduled by the EDF policy. This is done in two steps. First, the model is solved to optimize for processor utilization U (the rationale for choosing U is discussed in Section 5.2.3). Second, an algorithm is used to add scheduling constraints to the NLP model until a feasible EDF schedule is obtained or the problem is deemed infeasible.

5.2.1 Parameters

Parameters are the set of values that describe a particular system to be partitioned. The application task set is denoted τ. Each application task τ_i in τ has a set of nodes N_i and edges E_i. Each task also has deadline D_i and period T_i. The kernel has node set N_k but the edge set is not used because cut kernel edges have zero cost as discussed in Section 5.1.3.

Since application tasks can have nodes in common, the set of all unique application task nodes is defined: $N_\tau = \bigcup_{\tau_i \in \tau} N_i$. Nodes are not shared between the application tasks and the kernel: $N_\tau \cap N_k = \emptyset$. The set of all nodes N, is the union of task nodes and kernel nodes: $N = N_\tau \cup N_k$. The hardware size of node a in set N is $sz_a[hw]$ and the software size is $sz_a[sw]$. Node size is independent of the tasks or kernel to which it belongs. However, the same node may have different total worst-case execution times in different tasks because it may be invoked different numbers of times. Therefore, node worst-case execution time is defined per task or kernel. Task i node a has worst-case execution time in hardware, $c_{i,a}[hw]$, and software, $c_{i,a}[sw]$. Kernel node a has times $c_{k,a}[hw]$ and $c_{k,a}[sw]$. These times represent the total time spent executing the node during an invocation of τ_i or the kernel. (The Slif annotation gives node execution time per node execution, so a transformation is required to total execution time as described in Appendix Section B.1.)

The set of all edges is defined: $E = \bigcup_{\tau_i \in \tau} E_i$. Edge p in set E, e_p, has source node, $e_p[src]$, and destination node, $e_p[dst]$, and invocation count (f_p). The count represents the total invocations along the edge, per task instance. (The Slif edge count annotation represents the number of times that the destination node is invoked by the source node per execution of the source node. The transformation from the Slif edge count annotations to total edge count, as required for this model, is described in Appendix Section B.1.)

The implementation target is described implicitly by the application and kernel node worst-case execution times and sizes. The only other parameters required for the target are the programmable logic capacity, $Sz[hw]$ and memory capacity, $Sz[sw]$. Note that the processor capacity, cycles per second, is described implicitly in the node execution times. The schedule feasibility analysis checks that the processor capacity is not exceeded.

The nature of some nodes may require that they be bound to either hardware or software. For example the context save of the kernel must be executed on the processor. The set of nodes whose binding to hardware is predetermined is $B[hw]$ and to software is $B[sw]$.

Table 5.2, at the end of this chapter, contains a list of all parameters and variables used to define the NLP partitioning model and the heuristic partitioner of Chapter 6.

5.2.2 Variables

The following 0-1 variables have been used to formulate the NLP partitioning model. (A variable value 0 represents false and 1 true.)

Name	Description
$n_a[hw]$	node a is bound to hardware
$n_a[sw]$	node a is bound to software
$e_p[h/s]$	edge p is cut (source node in hardware and destination node in software)
$e_p[s/h]$	edge p is cut (source node in software and destination node in hardware)

5.2.3 Objective Function

The final goal of the partitioning is to pass the feasibility analysis of Algorithm 3.2. Algorithm 3.2 checks that processor utilization is not greater than one and then performs the processor demand analysis. The strategy of the partitioner is to first ensure that $U \leq 1$, and then to modify to partition to pass the processor demand analysis. The initial objective is therefore to minimize processor utilization:

$$\min U = \sum_{\tau_i \in \tau} \frac{C_i}{T_i},$$

where C_i is the worst-case execution time of task i. C_i is calculated:

$$\begin{aligned} C_i &= \sum_{n_a \in N_i} \left(c_{i,a}[hw] \cdot n_a[hw] + c_{i,a}[sw] \cdot n_a[sw]\right) \\ &\quad +2C_k \cdot n_{root}[sw] \\ &\quad +2C_k \sum_{e_p \in E_i} f_p \left(e_p[h/s] + e_p[s/h]\right), \end{aligned} \tag{5.1}$$

The first line sums worst-case execution times of τ_i nodes according to their bindings (hardware or software). The reason that both software and hardware times are included is that Algorithm 3.2 is used by the feasibility repair algorithm (Section 5.2.6), which calculates C_i by summing the worst-case execution times of nodes assigned to software and nodes assigned to hardware.

The second line adds $2C_k$ if the root node of τ_i is bound to software, where C_k is the kernel software worst-case execution time. The third line adds $2C_k$ for each cut edge, multiplied by the edge count. The rationale for adding these terms is described in Section 5.1.3. C_k is the sum of worst-case execution times of kernel nodes according to their binding (hardware or software):

$$C_k = \sum_{n_a \in N_k} \left(c_{k,a}[hw] \cdot n_a[hw] + c_{k,a}[sw] \cdot n_a[sw] \right). \tag{5.2}$$

The objective function for this NLP model is to minimize processor utilization. This optimization goal and others such as minimum execution time and minimum cut set are evaluated in Chapter 7 for their impact on schedule feasibility. The objective function is quadratic because the $n_a[hw|sw]$ variables are multiplied by the $e_p[h/s|s/h]$ variables when Equation 5.2 is substituted into Equation 5.1.

5.2.4 Constraints

1. Each node is assigned to either hardware or software.

   $\forall n_a \in N : \qquad n_a[hw] + n_a[sw] = 1.$

2. An edge is cut if the nodes that it connects have different bindings.

   $$\begin{aligned} \forall e_p \in E : \qquad & n_a[hw] + n_b[sw] - e_p[h/s] \leq 1, \\ & n_a[sw] + n_b[hw] - e_p[s/h] \leq 1, \end{aligned}$$

   where $n_a = e_p[src]$ and $n_b = e_p[dst]$.

   If edge e_p is cut, then one of $e_p[h/s]$ or $e_p[s/h]$ is set to one. These variables are are used in the calculation of C_i (Equation 5.1). Consider $e_p[h/s]$: if the source node is bound to hardware and the destination node is bound to software, it is set to one. If not, then optimization of the objective function sets $e_p[h/s]$ to zero.

3. Hardware and software capacities must not be exceeded.

   $\sum_{n_a \in N} sz_a[hw] \cdot n_a[hw] \leq Sz[hw]$.

   $\sum_{n_a \in N} sz_a[sw] \cdot n_a[sw] \leq Sz[sw]$.

   The first equation ensures that the sum of sizes of application and kernel nodes bound to hardware does not exceed $Sz[hw]$. The second equation performs the same check for software.

4. Some nodes may have predetermined bindings.

   $\forall n_a \in B[hw]: \quad n_a[hw] = 1.$

   $\forall n_a \in B[sw]: \quad n_a[sw] = 1.$

These constraints are a "knapsack" formulation and so the problem is a "0-1 quadratic knapsack" assignment problem [35].

5.2.5 Vectored Resource Capacities

The capacities of the hardware target, $Sz[hw]$, and the software target, $Sz[sw]$, have been modeled as unit capacities: representing FPGA logic elements or processor memory, for example. It may be desirable to associate multiple resource capacities with an implementation target. For example the FPGA could have logic element, embedded system block and pin capacities; or the processor could have separate program memory and data memory capacities. Or perhaps system level constraints such as design effort (time) could be added. Multiple resource capacities could be modeled using vectored capacity parameters, $\vec{Sz}[hw]$ and $\vec{Sz}[sw]$. The sizes of $\vec{Sz}[hw]$ and $\vec{Sz}[sw]$ need not be the same. The node size annotations would also be represented in vector form: $\vec{sz}_a[hw]$ and $\vec{sz}_a[sw]$. Their sizes would parallel the capacity vector sizes: $|\vec{sz}_a[hw]| = |\vec{Sz}[hw]|$ and $|\vec{sz}_a[sw]| = |\vec{Sz}[sw]|$. Constraint 3 would be repeated for each element in the vector.

5.2.6 Feasibility Repair Algorithm

When the NLP model is solved, processor utilization, U, is minimized and worst-case execution times are generated. Now one of Algorithm 3.2 or Algorithm 3.4 can be used to check whether the task set can be feasibly scheduled by the EDF policy. Algorithm 3.4 can only be used for task sets in which one task invokes coprocessors. Solutions produced by the NLP model may have multiple tasks that block on coprocessors, ruling out the use of Algorithm 3.4. Therefore Algorithm 3.2 is used to verify schedule feasibility.

Algorithm 3.2 checks that processor utilization does not exceed one: $U \leq 1$. Then it generates the set S of deadline events. The deadline events are checked in order. If for any deadline event $v \in S$ the processor demand exceeds the available time ($h(v) > v$), then the EDF feasibility analysis fails. However it might be possible to modify the partition to meet these deadlines. The set of deadline events, S, is returned by the analysis. The processor demand analysis at each event $v \in S$ is added to the NLP model as a fifth constraint:

5. At each deadline event, v in S, the processor demand $h(v)$ must not exceed v.
$$\forall v \in S : \qquad \sum_{D_i \leq v} \left(1 + \left\lfloor \frac{v - D_i}{T_i} \right\rfloor\right) C_i \quad \leq \quad v.$$

The NLP model is solved again with this new constraint added. Note that the NLP model is solved the first time without Constraint 5 because the output of the model (task worst-case execution time) is required to determine the synchronous busy period $[0, L)$ over which S is generated.

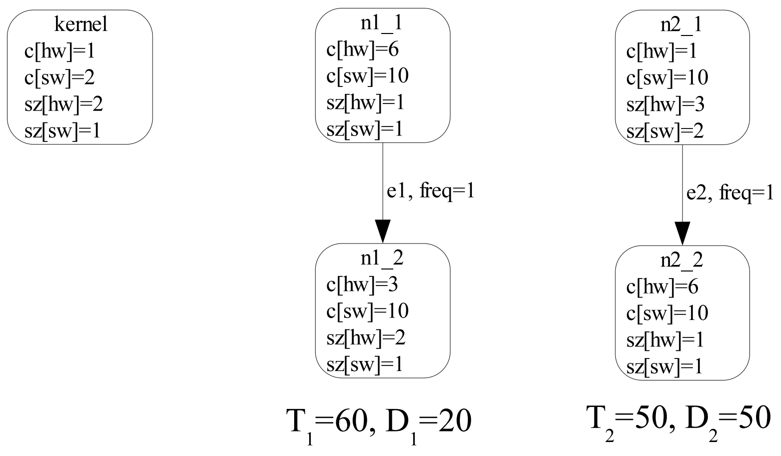
If a feasible solution to the revised model is found then the feasibility analysis must be repeated because L may have changed due to changed task execution times. A bigger L may add more deadline events that need checking. This process is repeated until the feasibility analysis passes or the NLP model becomes infeasible. The pseudocode for this feasibility repair algorithm is listed in Algorithm 5.1.

```
Result: partitioned application task set and kernel
solve NLP model (min. U with constraints 1 – 4) ;
while $U \leq 1$ do
    $S \leftarrow$ invoke Algorithm 3.2 ;
    if EDF analysis (Algorithm 3.2) passes then
        return solution ;
    end
    use $S$ to generate constraint 5 ;
    solve NLP model (min. $U$ with constraints 1 – 5) ;
    if “NLP model infeasible” then
        return no solution ;
    end
end
return no solution ;
```

Algorithm 5.1: Feasibility Repair Algorithm

Use of the feasibility repair algorithm is demonstrated using the task set and kernel represented in Figure 5.3. In this simple example, the kernel has one node (and no scheduling information). The two tasks consist of two nodes each with scheduling information as displayed. Hardware and software worst-case execution times and sizes are given for each node. The programmable logic capacity is $Sz[hw] = 2$ and the program/data memory capacity is $Sz[sw] = 10$.

When U is minimized, the kernel node is bound to hardware and all other nodes are



(a) kernel (b) task 1 (c) task 2

Figure 5.3: Feasibility Repair Example

bound to software, which corresponds to:

$$\begin{aligned} C_k &= 1 \\ C_1 &= 10 + 10 + 2C_k = 22 \\ C_2 &= 10 + 10 + 2C_k = 22 \\ U &= \frac{22}{60} + \frac{22}{50} = 0.8067. \end{aligned}$$

In calculating both C_1 and C_2, $2C_k$ is added to allow for task release/terminate. When Algorithm 3.2 is invoked, $L = 44$ and $S = \{20\}$. When the (only) deadline at $t = 20$ is checked, processor demand exceeds capacity ($h(20) = 22$) and the analysis fails.

A fifth constraint is added to the model to enforce that $h(20) \leq 20$. Since only τ_1 has a deadline by $t = 20$, the constraint is $C_1 \leq 20$, which when expanded becomes:

$$\begin{aligned}
20 \geq\ & (6n1\_1[hw] + 10n1\_1[sw]) + (3n1\_2[hw] + 10n1\_2[sw]) \\
& +2C_k n1\_1[sw] \\
& +2C_k(e1[h/s] + e1[s/h]).
\end{aligned}$$

Solving the model again, nodes $n1\_1$ and $n2\_2$ are bound to hardware and all other nodes are bound to software, which corresponds to:

$$\begin{aligned}
C_k &= 2 \\
C_1 &= 6 + 10 + 2C_k = 20 \\
C_2 &= 10 + 6 + 2C_k + 2C_k = 24 \\
U &= \frac{20}{60} + \frac{24}{50} = 0.8133.
\end{aligned}$$

When calculating C_1, the root node is bound to hardware and so $2C_k$ is only added for the cut edge. When calculating C_2, the root node is bound to software and the other node to hardware, so $2C_k$ is added for task release/terminate and for the cut edge. When Algorithm 3.2 is invoked, $L = 44$ and $S = \{20\}$. This time $h(20) = 20$ and the analysis passes.

5.3 Summary

A hardware/software partitioning problem has been defined that includes kernel and application tasks and that integrates EDF feasibility analysis of concurrent applications. The model is limited to SoCs/SoPCs with one processor. It is assumed that edges only incur additional cost if they are cut. An NLP model has been defined that minimizes processor utilization U, while accounting for extra kernel overhead to service cut edges in application task Slifs. A feasibility repair algorithm was also presented that adds constraints to the NLP model to ensure feasible scheduling by the EDF policy.

Table 5.2: Partitioning Symbols

Symbol	Indices	Definition
τ	i, j	task set
N	a, b	node set
N_τ, N_k	a, b	set of all unique task nodes, set of kernel nodes
N_i	a, b	task i node set
E	p, q	edge set
E_i	p, q	task i edge set
$E_{n_{i,a}}$	p, q	task i node a edge set
$e_p[src], e_p[dst]$		edge p source and destination nodes
f_p		edge p traversal count ("frequency")
$B[hw], B[sw]$	a, b	predetermined node binding sets
Hw, Sw		hardware and software partitions
$c_{i,a}[hw\|sw], c_{k,a}[hw\|sw]$		task i node a and kernel node a worst-case execution times
$sz_a[hw], sz_a[sw]$		node a hardware and software sizes
$Sz[hw], Sz[sw]$		target size limits
$n_a[hw], n_a[sw]$		node a binding variables
$e_p[h/s], e_p[s/h]$		edge p cut variables
U		processor demand
C, T, D	i, j	task worst-case execution time, period and deadline
χ	i	cutset
g_a^c		gain (execution time) of node a
g_a^χ		gain (cutset) of node a
g_a^U		gain (processor utilization) of node a
ν_k		kernel invocation frequency

Chapter 6

Heuristic Partitioning

The hardware/software partitioning problem was defined and an NLP model was developed in Chapter 5. In this chapter, a heuristic is proposed that solves the same problem as the NLP model. The goal in developing the heuristic is to find good solutions at a fraction of the run-time of the optimal NLP solution.

The heuristic is similar to the Fiduccia/Mattheyses (FM) node move heuristic [31] discussed in Chapter 2 except that gain for node moves is calculated differently. FM was chosen since its computational complexity is less than that of the Kernighan/Lin heuristic [54, 31]. Krishnamurthy's [58] or Sanchis' [88] heuristic were not used because the Slif graph edges only connect two nodes and so level gains are not applicable.

The heuristic pseudocode is shown in Algorithm 6.1. The heuristic starts with an (almost) arbitrary partition of the node set N into a set of hardware nodes, Hw, and a set of software nodes, Sw. Any nodes with predetermined bindings (i.e. belonging to $B[hw]$ or $B[sw]$) are permanently bound to the appropriate partition. The size limits of the hardware and software targets are never violated. The node move heuristic chooses the node with the largest gain of all the unlocked nodes that can be moved without violating size limits. Once moved, a node is locked and cannot move until the end of the pass. When a node is moved it changes the gains of some other nodes, so these are updated. Nodes continue to be moved until no unlocked nodes remain that can be moved without violating size limits. This ends the pass. The step s on which the maximum gain was achieved is selected as the starting point for the next pass. If the maximum gain of the pass was zero, then the partition is locally optimal and the heuristic stops.

Table 5.2, at the end of Chapter 5, is a list of all symbols used in defining this heuristic and in defining the NLP partitioning model.

6.1 Node Move Gains

The heuristic chooses a node to move based on its gain. As with the NLP model, the final goal of the partitioning is to pass the feasibility analysis of Algorithm 3.2. To pass the analysis, processor utilization must not exceed one, $U \leq 1$, and the processor demand must not exceed processor time at each deadline that is checked ($\forall v \in S : h(v) \leq v$). There are two goals to consider for the heuristic: 1) minimize U, and 2) minimize $h(v)$ for each v in S. Four gains are discussed below that may help meet one or both of these goals.

6.1.1 Worst-Case Execution Time Gain

The first and least complex to calculate gain is the decrease in worst-case execution time. This may aid scheduling by reducing processor demand before a deadline. Since an application task node can be shared by multiple tasks but a kernel node cannot be shared, task node gains and kernel node gains are calculated differently.

```
Data: $N = N_\tau \cup N_k$
      $Hw$ = hardware node subset, $Sw$ = software node subset
      $Sz[hw]$ = hardware Size Limit, $Sz[sw]$ = software Size Limit
start with any arbitrary partition $\{Hw, Sw\}$ such that:
  $\circ Hw \cup Sw = N$, $Hw \cap Sw = \emptyset$
  $\circ \sum_{a \in Hw} sz_a[hw] \leq Sz[hw]$ and $\sum_{a \in Sw} sz_a[sw] \leq Sz[sw]$
  $\circ \forall n_a \in B[hw] : n_a \in Hw$ and $\forall n_a \in B[sw] : n_a \in Sw$;
all nodes are unlocked (except those in $B[hw]$ and $B[sw]$);
repeat
    $s \leftarrow 1$ ;
    repeat
        choose 1 unlocked node to move such that:
          $\circ$gain $g$ is maximized
          $\circ Sz[hw]$ and $Sz[sw]$ are not violated;
        lock moved node;
1       update gains of other nodes;
        $s \leftarrow s + 1$ ;
    until until no nodes can move;
    choose step $s$ on which the sum of gains $G = \sum_{i=1}^{s} g_i$ is maximized ;
    if $G > 0$ then
        unlock all nodes (except those in $B[hw]$ and $B[sw]$);
        restore partition from step $s$;
    end
until until $G = 0$;
partition is locally optimal ;
```

Algorithm 6.1: Partitioning Heuristic

$$\text{If } n_a \in N_\tau : \quad \begin{aligned} g_a^c &= \sum_{\substack{\tau_i \in \tau \\ n_{i,a}=n_a}} g_{i,a}^c \\ g_{i,a}^c &= c_{i,a}[old] - c_{i,a}[new] \end{aligned} \tag{6.1}$$

$$\text{Else } (n_a \in N_k) : \quad g_a^c = c_{k,a}[old] - c_{k,a}[new] \tag{6.2}$$

Task node gain, g_a^c for $n_a \in N_\tau$, is the sum of worst-case execution time differences $g_{i,a}^c$ for each task τ_i in which n_a is used. $g_{i,a}^c$ is calculated by subtracting the worst-case execution time in the [*new*] partition (*hw* or *sw*) from the worst-case execution time in the [*old*] partition (*sw* or *hw*). Kernel node gain, g_a^c for $n_a \in N_k$, is not a sum since kernel nodes are not shared. For both task and kernel nodes, gain g_a^c does not change in a pass: it is independent of other nodes or edges. As a result no update of other node gains is required when a node is moved (Algorithm 6.1, Label 1).

6.1.2 Scaled, Weighted Cutset Gain

The original FM heuristic calculates gain as the decrease in cutset. As discussed in Section 5.1.3, cut application task edges incur an additional cost of two kernel invocations. Reducing the number of kernel invocations contributes to a decrease in processor utilization. The cutset is weighted by the edge invocation count parameter which is described in Section 5.2.1. Since different tasks can have different frequencies, the task edge weights (counts) are scaled by task frequency. The scaled, weighted cutset is denoted χ.

$$\text{If } n_a \in N_\tau : \quad \begin{aligned} g_a^\chi &= \sum_{\substack{\tau_i \in \tau \\ n_{i,a}=n_a}} \frac{g_{i,a}^\chi}{T_i} \\ g_{i,a}^\chi &= \sum_{\substack{e_p \in E_{n_{i,a}} \\ n_b \in e_p \\ n_b[new]=1}} f_p - \sum_{\substack{e_p \in E_{n_{i,a}} \\ n_b \in e_p \\ n_b[old]=1}} f_p \end{aligned} \tag{6.3}$$

$$\text{Else } (n_a \in N_k) : \quad g_a^\chi = 0 \tag{6.4}$$

Task node gain g_a^χ sums the weighted cutset gains, divided by task period for each task to which n_a belongs. The gain $g^{\chi}_{i,a}$ of moving task i node a from $[old]$ to $[new]$ is calculated by summing the cut edge weights and subtracting the sum of uncut edge weights. Set $E_{n_{i,a}}$ is the set of edges in task i to which node n_a is connected. Node n_b is the node to which node n_a is connected by edge $e_p \in E_{n_{i,a}}$. Those edges connected to nodes in $[new]$ will become uncut by the move and their edge weight (count) adds to gain. Those edges connected to nodes in $[old]$ will become cut and subtract from gain. Kernel edges are assigned zero weight since they are assumed to incur no additional cost when cut (discussed in Section 5.1.3). When task node n_a is moved, the gains of other nodes that connect to it must be updated as described in Algorithm 6.2.

```
foreach $\tau_i \in \tau$ do
    foreach edge $e_p \in E_{n_{i,a}}$ do
        $n_b$ is other node on $e_p$ ;
        if binding of $n_b$ = new binding of $n_a$ then
            $\Delta g^{\chi}_{i,b} = -2f_p$ ;
        else
            $\Delta g^{\chi}_{i,b} = +2f_p$ ;
        end
        $g^{\chi}_b = g^{\chi}_b + \frac{\Delta g^{\chi}_{i,b}}{T_i}$ ;
    end
end
```

Algorithm 6.2: χ Gain Update

6.1.3 Processor Utilization Gain

A more direct way to ensure that U does not exceed one is to define gain as the decrease in U. However this is a more complex gain calculation and more work is required to update other node gains when a node is moved. The gain in processor utilization g^n_a when node n_a is moved incorporates the two previous gains, g^c and g^χ.

$$\begin{aligned}
\text{If } n_a \in N_\tau : \quad g^U_a &= \sum_{\substack{\tau_i \in \tau \\ n_{i,a} = n_a}} \frac{g^U_{i,a}}{T_i} \\
g^U_{i,a} &= \text{gain in } C_i = g^c_{i,a} + 2C_k\left(g^\chi_{i,a}\right) \\
&\quad \text{if } n_a = n_{i,root} \text{ and } [new] = [hw] : g^U_{i,a} = g^U_{i,a} + 2C_k \\
&\quad \text{else if } n_a = n_{i,root} \text{ and } [new] = [sw] : g^U_{i,a} = g^U_{i,a} - 2C_k
\end{aligned} \tag{6.5}$$

$$\begin{aligned}
\text{Else } (n_a \in N_k) : \quad g^U_a &= \nu_k \cdot g^c_a \\
\nu_k &= \sum_{\tau_i \in \tau} \frac{2(\chi_i + n_{i,root}[sw])}{T_i}
\end{aligned} \tag{6.6}$$

The gain of moving task node n_a is the sum of changes in C_i divided by T_i for each task which uses n_a. C_i is calculated according to Equation 5.1 which is reflected in the calculation of $g^U_{i,a}$. There are three parts:

1. decrease in node execution time - g^c_{ia},

2. decrease in cutset - $g^\chi_{i,a}$ (each cut edge incurs two kernel invocations - C_k), and

3. if the node is the root of the Slif and it is moved to software, $2C_k$ is added for release/terminate, decreasing gain, and if it is moved to hardware $2C_k$ is subtracted, increasing gain (as discussed in Section 5.1.3).

The gain of moving kernel node n_a is the product of decreased node execution time g^c_a and kernel invocation frequency ν_k. The kernel is invoked by each task's cutset (χ_i = sum of task i cut edge counts) and once more if the task's root node is in software ($n_{i,root}[sw] = 1$). Each task's kernel invocation count is divided by its period and summed to form ν_k, the kernel invocation frequency.

The gains of kernel nodes and task nodes are now inter-connected. If a kernel node is moved, C_k changes, which changes task node gains, so all of the task node gains must be recalculated. If a task node is moved, that task's cutset changes, which changes ν_k, so all of the kernel node gains must be recalculated. It is because of the inter-connected nature of task and kernel node moves that tasks may share nodes with each other but not with the kernel (as stated in Section 5.1.1). The gain update algorithms are described in Algorithms 6.3 and 6.4.

Algorithm 6.3 updates node gains when a task node is moved. The gains of other task nodes that are connected to it by edges change because the edges becomes either cut or uncut. This also affects the frequency with which the kernel is invoked, ν_k. After updating gains of all connected task nodes, all kernel node gains are updated with the changed ν_k.

Algorithm 6.4 updates node gains when a kernel node is moved. The other kernel nodes are not affected since cut kernel edges do not incur a penalty. However, C_k changes which requires that all task node gains are updated.

6.1.4 Gain Summary

Three node move gains have been proposed to improve the partitioned task set EDF schedule feasibility. The gains, in order of calculation complexity (and expected solution quality), are:

1. g^c – decrease in node worst-case execution time,
2. g^χ – decrease in scaled, weighted cutset,
3. g^U – decrease in processor utilization, and
4. g^U/sz – decrease in processor utilization divided by node hardware size.

A fourth gain has been added that is related to ΔU/LE metric suggested in Chapter 4: g^U/sz. It is expected that the hardware size limit will restrict the solution space more

```
foreach $\tau_i$ such that $n_{i,a} = n_a$ do
    foreach edge $e_p \in E_{n_{i,a}}$ do
        $n_b$ is other node on $e_p$ ;
        if binding of $n_b$ = new binding of $n_a$ then
            $\Delta g^U_{i,b} = -2\,(2C_k \cdot f_p)$ ;
            $\Delta \nu_k = -2 \cdot f_p$ ;
        else
            $\Delta g^U_{i,b} = +2\,(2C_k \cdot f_p)$ ;
            $\Delta \nu_k = +2 \cdot f_p$ ;
        end
        $g^U_b = g^U_b + \frac{\Delta g^U_{i,b}}{T_i}$ ;
        $\nu_k = \nu_k + \frac{\Delta \nu_k}{T_i}$ ;
    end
    if $n_a$ is the root node and it is now in software then
        $\nu_k = \nu_k + \frac{2}{T_i}$ ;
    else if $n_a$ is the root node and it is now in hardware then
        $\nu_k = \nu_k - \frac{2}{T_i}$ ;
    end
end
foreach $n_a \in N_k$ do
    recalculate $g^U_a$ using Equation 6.6;
end
```

Algorithm 6.3: U Gain Update (Task Node Move)

```
$C_k = C_k + g_a^c$ ;
foreach $n_a \in N_\tau$ do
    recalculate $g_a^U$ using Equation 6.5;
end
```

Algorithm 6.4: U Gain Update (Kernel Node Move)

than the software size limit. Since the heuristic chooses the largest gain possible in each step, it may fill up the hardware with a small number of nodes with large gain. Better results might be obtained if the node's gain is scaled by its size so that the hardware does not get dominated by a few nodes. The results comparing these four gains are presented in Chapter 7.

6.2 Feasibility Repair Heuristic

As with the NLP model of Chapter 5, after executing the heuristic partitioner, the solution may require repair to pass the EDF feasibility analysis of Algorithm 3.2. The repair heuristic, which is based on the FM heuristic, is described in Algorithm 6.5. If $U > 1$ then the solution cannot be repaired. If $U \leq 1$ but the task set fails the EDF analysis of Algorithm 3.2, then the heuristic attempts to fix missed deadlines, one at a time. The processor demand at the first missed deadline v_k is selected as the goal for the partitioner. Recall that processor demand is calculated:

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right) C_i.$$

Node gain g_a^h is defined as the decrease in $h(v_k)$. It is calculated by summing the decrease that moving node n_a causes to each C_i multiplied by the number of times that the C_i is summed in the $h(v_k)$ calculation. The heuristic finds a locally optimal processor demand at the missed deadline (minimizes $h(v_k)$). At the same time the heuristic is restricted so that it does not cause earlier deadlines to be violated by a node move (i.e. a node move must satisfy the constraint $\forall v_j < v_k : h(v_j) \leq v_j$). After minimizing processor demand at the violated deadline, the relation $h(v_k) \leq v_k$ is tested. If false, then the solution cannot be repaired. If true, then the next missed deadline is selected for repair until no deadlines are missed or a deadline is found that cannot be met.

6.3 Summary

A heuristic solution to the hardware/software partitioning problem has been proposed. It is based upon the Fiduccia/Mattheyses heuristic which is used for netlist partitioning in VLSI circuit layout. Four gain metrics have been proposed: g^c - worst-case execution time, g^χ - scaled, weighted cutset, g^U - processor utilization, and g^U/sz - processor utilization divided by size. Results from comparing these gains are presented in Chapter 7. The performance and solutions of the NLP model and the heuristic are also compared in Chapter 7. A heuristic has also been proposed to repair the heuristic partitioning solution so that it can be feasibly scheduled by the preemptive EDF policy.

```
Input: data from partitioning heuristic
Result: partition with feasible schedule
while $U \leq 1$ do
    if EDF analysis (Algorithm 3.2) passes then
        return solution ;
    end
    select first violated deadline $v_k$ (s.t. $h(v_k) > v_k$) ;
    all nodes are unlocked (except those in $B[hw]$ and $B[sw]$);
    repeat
        $s \leftarrow 1$ ;
        repeat
            choose 1 unlocked node to move such that:
                ○ gain $g = \Delta h(v_k)$ is maximized
                ○ no deadlines before $v_k$ are missed ($\forall v_j < v_k : h(v_j) \leq v_j$)
                ○$Sz[hw]$ and $Sz[sw]$ are not violated;
            lock moved node;
            update gains of other nodes;
            $s \leftarrow s + 1$ ;
        until until no nodes can move;
        choose step $s$ on which the sum of gains $G = \sum_{i=1}^{s} g_i$ is maximized ;
        if $G > 0$ then
            unlock all nodes (except those in $B[hw]$ and $B[sw]$);
            restore partition from step $s$;
        end
    until until $G = 0$;
    if $h(v_k) > v_k$ then
        return no solution ;
    end
end
return no solution ;
```

Algorithm 6.5: Feasibility Repair Heuristic

Chapter 7

Results

In this chapter results are presented for the NLP model and heuristic partitioners of Chapters 5 and 6. Two data sources were used to test the partitioners. The first data source is the case study of the idle engine application. The case study provides one test case with both scheduling parameters and measurable node execution times in software. It's limitation is a shortage of measurable node execution times and sizes in hardware. The method used to estimate these parameters is discussed in Section 7.1. The second data source is generic test cases generated automatically by the gpslifgen tool (General Purpose Slif Generator) [97]. The generic test cases have node execution times and sizes in both software and hardware. However they lack scheduling parameters. The method used to generate the scheduling parameters is described in Section 7.2. Before comparing the heuristic results against the NLP model results, the number of times to run the heuristic per test case must be determined. The heuristic starts each run with a random partition and proceeds to find the locally optimal solution. In Section 7.3 the number of runs of the heuristic per test case is determined experimentally. The results from comparing the NLP model with the heuristic partitioner are presented in Section 7.4. The optimal partition, obtained by solving the NLP model, for the idle engine test case is examined in detail in Section 7.4.1. In Section 7.5, results are presented from the evaluation of the four gains that were proposed for the heuristic in Chapter 6.

7.1 Case Study Data

In Chapter 4 a case study of an SoPC is presented. The application consists of six tasks scheduled by a multi-tasking kernel. To model the test case for hardware/software partitioning, a Slif graph was created for each of the application tasks and for the kernel. These Slif graphs are depicted in Appendix Figures B.3 – B.9. The graph annotations required by the partitioners are:

1. edge invocation count,
2. node hardware size (logic elements),
3. node hardware worst-case execution time (cycles), and
4. node software worst-case execution time (cycles).

Also associated with each task is a deadline and period as listed in Figure 4.4.

Before discussing the four Slif graph annotations above, the implementation targets need comment. The two implementation targets in the case study were the programmable logic of the FPGA and the processor. Two FPGA resource constraints were discussed in the case study: logic elements (LE) and embedded system blocks (ESB). It was shown in Section 4.5.1 that LEs were the limiting factor in fitting the coprocessors onto the FPGA. Therefore the number of LEs have been used as the measure of hardware size in the Slif graph node annotations. The hardware size constraint, $Sz[hw]$, was 4914 LEs, the number of LEs not used by the processor and standard peripherals (UART, timer).

The two software resource constraints were memory and processor cycles. Cycles are counted in the node software worst-case execution time annotation and accounted for by the EDF schedule feasibility analysis. The development board provides 256KB of SRAM of which the total application only needs 40KB for program memory and 24KB for task stacks. Therefore no limit on software size, $Sz[sw]$, was set and node software size annotations were not used.

The four required Slif graph annotations are now discussed. Edge invocation counts were produced by a manual examination of the application and kernel source code. (The application and kernel were coded in C++). Node software worst-case execution times were produced in one of two ways.

1. Where function source code was available, the compiler was used to produce assembly listings from which worst-case execution times were manually calculated.
2. Source code was not available for C++ library functions such as the cos() function. In this case, the application was executed for 10 seconds and a timer was used to capture node worst-case execution time.

Node hardware worst-case execution time and node hardware size were only measurable for the cos node of the Environment task. A coprocessor was implemented that calculated cosine by the cordic algorithm (Section 4.3). A method was needed to estimate node hardware worst-case execution and node hardware size for all the other case study nodes, based on the only node data available: software worst-case execution time. A list of parameters that are known and those that need estimates is given in Table 7.1 (which also lists the GPS1 and GPS2 test cases described in Section 7.2). Those that are known have a $\surd$, otherwise they need to be estimated.

Table 7.1: Known and Estimated Parameters

	$c[sw]$	$c[hw]$	$sz[hw]$	D	T	$Sz[hw]$
Idle Eng. - cos()	$\surd$	est.	est.	$\surd$	$\surd$	$\surd$
cos()	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
GPS1	$\surd$	$\surd$	$\surd$	est.	est.	est.
GPS2	$\surd$	$\surd$	$\surd$	est.	est.	est.

Note that another coprocessor was implemented to replace a part of the kernel (Section 4.4). However the cs2 coprocessor does not fit well with the kernel Slif graph because the coprocessor replaces two whole nodes (MinHeap::pop and MinHeap::insert) and parts of the Sched::doSched function. This limitation gives reason to consider another representation scheme such as control and data-flow graphs (CDFG) that represent at the basic block level. However, as discussed in Section 7.2, a generic test case generator was available whose output was Slif graphs. Therefore the choice of the Slif representation was a trade-off between flexibility of representation and availability of test cases.

7.1.1 Node Hardware Worst-Case Execution Time Estimation

The work of Vahid and dm Le [97] investigated the relationship between software execution time and hardware execution time. They performed a linear regression using data from their six benchmark applications. The linear regression parameters are reproduced in Table 7.2.

The ratio of hardware execution time to software execution time, $R^c = \frac{c[hw]}{c[sw]}$, ranges from 0.0188 to 0.0406. In contrast, the ratio for the cos function of the case study was $R^c = \frac{2700}{47961} = 0.0563$. The results of an experiment to choose a ratio for the idle engine test case are reported in Section 7.1.3. Having proposed a method to estimate node hardware

Table 7.2: Replication of Table 1 from [97]

Bench- mark	Nodes	Regression equation	Correlation coefficient
ANS	44	$c[hw] = 1.12 + 0.0404c[sw]$	+0.58
ETHER	124	$c[hw] = 0.228 + 0.0220c[sw]$	+0.92
FUZZY	69	$c[hw] = 0.24 + 0.0191c[sw]$	+0.98
ITV	84	$c[hw] = -0.070 + 0.0406c[sw]$	+0.89
MWT	31	$c[hw] = 0.462 + 0.0188c[sw]$	+0.83
VOL	38	$c[hw] = 0.245 + 0.0330c[sw]$	+0.78

execution time, node hardware size needs to be estimated.

7.1.2 Node Hardware Size Estimation

There are two numbers from which the node hardware size can be estimated: node software worst-case execution time and node hardware worst-case execution time. Node hardware size estimation is a very difficult problem and not within the scope of this research. However, a relationship must be chosen out of necessity. Appendix section B.2.1 describes the justification for the ratios used. The two ratios of hardware size to execution time, $R^{sz} = \frac{sz[hw]}{c[hw]}$, that were used were:

1. $R^{sz} = 4.74\text{x}10^7$ LE/s, and
2. $R^{sz} = 5.47\text{x}10^8$ LE/s.

The experiment reported in the next section was performed to choose the two ratios: R^c and R^{sz}.

7.1.3 Case Study Estimation Parameter Experiment

An experiment was conducted to help choose ratios R^c and R^{sz} by evaluating their impact on partitioning results. Three R^c values were tested.

- 0.0191 - This value is from line 3 of Table 7.2 was chosen for the high correlation coefficient.
- 0.0563 - This value is calculated from the cos() function and Cordic coprocessor data.
- 0.0376 - This value is the mid-point between the other two values.

Three R^{sz} values were tested.

- 4.74x10$^7 LE/s$ - This value is from the Cordic coprocessor.
- 5.47x10$^8 LE/s$ - This value is from the cs2 coprocessor.
- 2.48x10$^8 LE/s$ - This value lies between the other two values.

Node hardware worst-case execution time and size estimates are adjusted by an amount picked from the uniform random distribution $[+20\%, -20\%]$. For example:

$$sz[hw] = c[sw] \cdot R^{sz} \cdot (0.8 + 0.4r)\,,$$

where r is chosen from the random uniform distribution $[0, 1]$.

This experiment (as well as all other partitioning experiments) was conducted on a Intel Pentium-4 2.60GHz processor running the Linux operating system. The NLP model was solved using Opbdp [13], a Davis-Putnam based enumeration algorithm that finds an optimal solution. The heuristic was coded in C++.

Both the NLP model and the heuristic were used for this experiment. The NLP model was solved once for each R^c/R^{sz} combination. The heuristic was run 100 times using gain

Table 7.3: Idle Engine Estimation Parameter Results

	$R^c = \frac{c[hw]}{c[sw]}$		
$R^{sz} = \frac{sz[hw]}{c[hw]}$	0.0191	0.0376	0.0563
4.74x10^7 LE/s	opt U = 0.09174 # feas = 99 min U = 0.09174 (+0%) avg U = 0.1349 (+47%)	opt U = 0.1550 # feas = 100 min U = 0.1550 (+0%) avg U = 0.1649 (+6%)	opt U = 0.2161 # feas = 99 min U = 0.2161 (+0%) avg U = 0.2572 (+19%)
2.48x10^8 LE/s	opt U = 0.4360 # feas = 71 min U = 0.4360 (+0%) avg U = 0.4434 (+2%)	opt U = 0.4657 # feas = 31 min U = 0.4657 (+0%) avg U = 0.4807 (+3%)	opt U = 0.6722 # feas = 6 min U = 0.8018 (+19%) avg U = 0.8076 (+20%)
5.47x10^8 LE/s	opt U = 0.4713 # feas = 37 min U = 0.4713 (+0%) avg U = 0.4809 (+2%)	opt U = infeas. # feas = 0 min U = — avg U = —	opt U = infeas. # feas = 0 min U = — avg U = —

g^U for each combination. The optimal processor utilization (opt U) from the NLP model is reported in Table 7.3. The number of feasible solutions (# feas) and minimum (min U) and average (avg U) processor utilization for feasible solutions from the heuristic are also reported in Table 7.3. The difference between the optimal U and minimum U is reported as a percentage, as is the difference between the optimal U and average U. All cases except two in the third row were feasible when solved using the NLP solver, Opbdp. This can be explained by noting that as R^{sz} increases, the problem becomes more difficult because node hardware sizes increase but the hardware capacity, $Sz[hw]$, remains fixed.

Examining Table 7.3 by row, the first row shows that every run of the heuristic produced a feasible solution, while the last row results in 2 unsolvable test cases. The second row ($R^c = 2.48x10^8$ LE/s) was chosen for test cases that are more challenging than in the first row, but are still solvable, unlike the those in the third row. The second column of the second row ($R^{sz} = 0.0376$) was chosen for similar reasons. Also, it lies in the range of ratios presented in Table 7.2. Using these two ratios, the first partitioning data source, the idle engine test case, was prepared for partitioning. The estimated annotation values are shown in the Slif graphs in Appendix Figures B.3 – B.9.

7.2 Generic Problem Sets

To better evaluate the partitioners, a second data source was needed. Thus the gpslifgen tool [97] was used to automatically generate generic test cases of varying size. Two sets of eight generic test cases each were generated. The set of larger test cases, Generic Problem Set 1 (GPS1), was used as reported in Section 7.5 to test the various gain metrics proposed for the partitioning heuristic. It was found that the run-times of the NLP model solver, Opbdp, were too long on the GPS1 test cases. Therefore a set of smaller test cases, Generic Problem Set 2 (GPS2), was used to test the NLP model as reported in Section 7.4. The development of the set of larger test cases, GPS1, is explained in detail, followed by a short discussion of the set of smaller test cases, GPS2.

Vahid and dm Le [97] developed a method to automatically produce generic Slif graphs of arbitrary size (i.e. node count). The graph characteristics are based on statistics gathered from six real benchmarks. The four annotations described in Section 7.1 are present in the Slif graphs generated by gpslifgen: edge invocation count, node hardware size, node hardware worst-case execution time, and node software worst-case execution time. Ten percent of nodes in the generated Slif graphs are root nodes (a node with no incoming edges). Each root node is the root of a directed acyclic graph (DAG) which can be used as the Slif graph of an application task. The DAGs have nodes in common, so the tasks can have node counts ranging between one and ninety percent of the node count of the Slif graph generated by gpslifgen. Thus the one generated Slif graph can be interpreted as multiple single-root Slif graphs, each representing a task in a set of concurrent tasks.

Once a single-root Slif graph has been associated with a task, a deadline and period must be generated for it (see list of parameters to estimate in Table 7.1). In addition to generating task deadlines and period, an additional Slif graph is generated for the kernel and the size capacity of the hardware target is determined.

For each test case, a different kernel Slif graph is generated that has 11 nodes (this is the number of kernel nodes in the case study - Figure B.3). When gpslifgen generates the kernel Slif graph, it has two root nodes. This problem is corrected by adding an edge from one of the root nodes to the other. The edge has an invocation count of one. In the case study, five of the eleven kernel nodes are bound to software (e.g. the context switch nodes, _ldProcCtxt and _stProcCtxt, must be in software). This is imitated in the generated kernel Slif graph by determining randomly with a 5/11 probability if each node is bound to software.

Each task requires a deadline D_i and period T_i. In order to generate D_i and T_i, the task worst-case execution time C_i is first calculated. C_i is calculated using Equation 5.1 and assuming that all task nodes are bound to software. The magnitude of C_i is used as a reference for the relative size of D_i and T_i. This is done by choosing D_i from the uniform random distribution in $[w \cdot C_i, n \cdot C_i]$ where w is determined experimentally and n is the number of tasks. T_i is chosen from the uniform random distribution in $[x \cdot D_i, n \cdot C_i]$ where x is determined experimentally. These ranges for D_i and T_i are illustrated in Figure

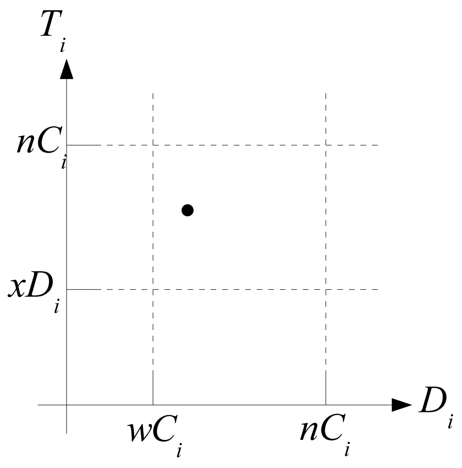


Figure 7.1: D_i and T_i Ranges

7.1. A point is being chosen randomly from within the boxed region. As D_i increases, the slack increases, making it easier to meet the deadline. As T_i increases, the task's processor utilization decreases, making it easier to schedule the task set. The value $n \cdot C_i$ was used as the upper bound for D_i and T_i because it results in a processor utilization of one, $U = 1$. If all task periods are equal to $n \cdot C_i$, then:

$$\begin{aligned} U &= \sum_{i=1}^{n} \frac{C_i}{T_i} \\ &= \sum_{i=1}^{n} \frac{C_i}{nC_i} \\ &= \sum_{i=1}^{n} \frac{1}{n} = 1 \end{aligned}$$

Recall that C_i is calculated assuming an all-software implementation. Therefore, in all but the special case where all $T_i = n \cdot C_i$, the test case will need partial implementation in hardware to satisfy the condition necessary for EDF scheduling that $U \leq 1$.

Schedule parameters w and x described above were tested by examining their effect on number of feasible solutions found by the heuristic for each generated test case. At the same time, an appropriate hardware size capacity $Sz[hw]$ was also investigated. The node sizes of all task and kernel nodes is summed to form the hardware total, hwTotal. $Sz[hw]$ is chosen from the uniform random distribution in $[y \cdot \text{hwTotal}, z \cdot \text{hwTotal}]$. The experiment with scheduling parameters w and x, and size parameters y and z is described next.

7.2.1 GPS1

Eight test cases are generated for GPS1. Two Slif graphs for each size (number of nodes) of 100, 200, 500 and 1000 were generated with gpslifgen. The test cases are labeled p100_1, p100_2, p200_1, etc. Nine combinations of schedule parameters were tested using $w = \{1, 2, 4\}$ and $x = \{1, 2, 4\}$. These nine combinations of schedule parameters where tested using two size ranges: $y = 0.01, z = 0.05$ and $y = 0.05, z = 0.10$. For each case (w, x, y, z), the eight test cases were formed and tested with the heuristic partitioner. The heuristic was invoked five times for each of the four proposed gain metrics (Chapter 6). Note that for a given combination of w, x, y, z, the values of D_i, T_i and $Sz[hw]$ do not change between runs of the heuristic partitioner. The results for size range $y = 0.01, z = 0.05$ are shown in Table 7.4 and the results for size range $y = 0.05, z = 0.10$ are shown in Table 7.5. If at least one feasible solution was found for a test case from the set, the table was marked with a $\surd$. Otherwise it was left unmarked.

Based on the results in Tables 7.4 and 7.5, schedule parameters $(w = 2, x = 2)$ and size parameters $(y = 0.01, z = 0.05)$ were selected for generating GPS1. The goal was to choose parameters that make the test cases difficult to solve, without generating too many unsolvable test cases. For the chosen parameters, six of eight generated test cases were solvable by this test. Problems p100_1 and p1000_1 were not solvable by this test with

Table 7.4: Problem Generation Parameters w, x $(y, z = 0.01, 0.05)$

$y - z$	0.01 – 0.05								
w	1			2			4		
x	1	2	4	1	2	4	1	2	4
p100_1								√	√
p100_2					√	√	√	√	√
p200_1		√	√	√	√	√	√	√	√
p200_2			√		√	√	√	√	√
p500_1		√			√	√	√	√	√
p500_2	√	√	√	√	√	√	√	√	√
p1000_1									
p1000_2	√	√	√	√	√	√	√	√	√

Table 7.5: Problem Generation Parameters w, x $(y, z = 0.05, 0.10)$

$y - z$	0.05 – 0.10								
w	1			2			4		
x	1	2	4	1	2	4	1	2	4
p100_1								√	√
p100_2					√	√	√	√	√
p200_1	√	√	√	√	√	√	√	√	√
p200_2			√		√	√	√	√	√
p500_1	√	√	√	√	√	√	√	√	√
p500_2	√	√	√	√	√	√	√	√	√
p1000_1									
p1000_2	√	√	√	√	√	√	√	√	√

these parameters. For these test cases, the following parameters were found to result in some feasible solutions:

- **p100_1** $(w = 4, x = 4)$ and $(y = 0.10, z = 0.20)$

- **p1000_1** $(w = 75, x = 75)$ and $(y = 0.10, z = 0.20)$

In both cases the hardware range was increased from $(y = 0.01, z = 0.05)$ to $(y = 0.10, z = 0.20)$, at least doubling the hardware target capacity. This gives the partitioner more choices of nodes to move between partitions. The schedule parameters were also increased with the result that the processor utilization for each was $U = 1$. In the testing results that follow (Sections 7.3 and 7.5), the results for these two test cases do not stand out from the others.

Table 7.6 gives a brief summary of the eight test cases generated using the parameters determined above. For each test case three values are reported:

1. number of tasks,
2. processor utilization (when all nodes in software), and
3. hardware target capacity as a percent of hwTotal.

In all cases but p100_1 and p1000_1 the processor utilization is greater than one indicating that the test case is infeasible unless some nodes are partitioned to hardware. For p100_1 and p1000_1, processor utilization is one but this does not necessarily mean that the test case is feasible: the task set must also pass the processor demand part of the EDF feasibility analysis (see Algorithm 3.2). Additional statistics regarding the demand on the processor

Table 7.6: Generated Problem: Summary Statistics

Problem	N Tasks	U (all sw)	Hw Size Frac
p100_1	10	1.000	15.3%
p100_2	10	1.040	2.53%
p200_1	20	1.228	1.12%
p200_2	20	1.095	1.19%
p500_1	50	1.197	4.39%
p500_2	50	1.147	2.26%
p1000_1	100	1.000	11.6%
p1000_2	100	1.132	1.25%

by the test cases is described in Appendix Table B.4 where C_i/T_i and C_i/D_i are reported for each test case.

7.2.2 GPS2

When the test cases in GPS1 were solved for the NLP model using Opbdp, the run-times exceeded 10000 seconds. (The run-times with 30 application nodes approaches 10000 seconds as reported in Table 7.8. Since the 0-1 knapsack problem is NP-hard [35], this time for 30 application nodes will grow exponentially for larger problems.) Therefore a second set of generic test cases, GPS2, was developed. The same methodology was used to make GPS2 as was used for GPS1. The main difference is that for all test cases, gpslifgen, was used to make generic Slif graphs of 30 nodes. These were used to make task sets with three tasks. The kernel Slif graphs were generated in the same manner as for GPS1. The schedule parameters remained the same ($w = 2, x = 2$) but the hardware size parameters were changed to ($y = 0.20, z = 0.50$). The bigger hardware allocation was needed to

Table 7.7: Percent Feasible Runs vs. # Runs

	# Runs				
Problem	25	50	100	200	400
idle eng	32	32	31	24.5	25.25
p100_1	72	82	82	83	85.75
p100_2	100	100	100	100	100
p200_1	88	92	92	93	95.25
p200_2	100	100	100	100	100
p500_1	4	4	4	4.5	3
p500_2	100	100	100	99	99.5
p1000_1	72	70	68	71	70
p1000_2	100	100	100	100	100

generate feasible test cases. The test cases in GPS2 are named p30_1, p30_2, … p30_8.

7.3 Heuristic Run Length

Before evaluating the heuristic partitioner, it is necessary to determine an appropriate number of runs ("run length"). For each run, the heuristic starts with a random solution and performs repeated node moves to find the local optimum. The heuristic is run multiple times per test case with a different random partition each time in an attempt to find a good solution. Five run lengths were tested: 25, 50, 100, 200 and 400. The idle engine test case and the eight test cases of GPS1 were solved with the heuristic using gain metric g^U. The number of feasible solutions found is recorded as a percent of the run length in Table 7.7.

For each test case, the percent of feasible solutions generated does not vary more than a few percent between runs of 50 and 400. The only exception is the idle engine test case where the percent of feasible solutions decreases by 6.75%. Run lengths of 100 are used for the results reported in the rest of this chapter.

7.4 NLP Model and Heuristic Comparison

This first set of results is from a test comparing the optimal NLP solution against the heuristic solution. The experimental platform was described in Section 7.1.3. The test data consisted of the idle engine test case and the eight test cases of GPS2. The NLP solver was run once to obtain the optimal solution. Run-time (in seconds) and optimal U were recorded for the NLP solver. The heuristic was run 100 times per test case and the total run-time (in seconds) was recorded. The number of feasible solutions and the minimum, average and standard deviation of U for feasible solutions were recorded. The number of times that the minimum U was found was also reported. These results are presented in Table 7.8.

The heuristic run times are 3 orders of magnitude faster than the NLP solver. The heuristic found the optimal solution at least once in every case. The heuristic found a feasible solution between 28% and 89% of the time. The standard deviation in processor utilization was small for seven cases (actually zero for five cases) indicating that there may have been few good solutions and that the heuristic found them most of the time. Two cases (p30_4 and p30_8) had larger variance, with average processor utilization at least double the optimal. However the optimal value was still found at least 23 times out of 100 for these two cases. The heuristic compared well with the NLP model for these relatively small test cases.

Table 7.8: NLP vs Heuristic Results

	NLP		Heuristic					
Task Set	Time	U	Time	# Feas	U_{min}	U_{avg}	U_{stddev}	# Min
idle eng	134.3	0.4657	0.12	31	0.4657	0.4807	0.01881	18
p30_1	8296.25	0.4358	0.14	45	0.4358	0.5020	0.04280	5
p30_2	2556.19	0.1643	0.16	28	0.1643	0.1643	0	28
p30_3	569.85	0.9824	0.10	66	0.9824	0.9824	0	66
p30_4	329.84	0.1933	0.19	82	0.1933	0.4292	0.3065	29
p30_5	431.56	0.7475	0.10	88	0.7475	0.7475	0	88
p30_6	3466.19	0.6010	0.11	32	0.6010	0.6010	0	32
p30_7	328.62	0.8022	0.15	46	0.8022	0.8022	0	46
p30_8	317.62	0.1418	0.13	89	0.1418	0.7319	0.3774	23

7.4.1 Analysis of Idle Engine Results

The partition found by the NLP model for the idle engine test case is compared here with the manual partitioning done in the case study of Chapter 4. The NLP model results are described in detail in Appendix Section B.2.3. The majority of nodes were assigned to hardware: 28 out of 41 nodes. Of the six kernel nodes not pre-bound to software, five were assigned to hardware. Notably, the cos() function was not bound to hardware as was done in the case study. The processor utilizations achieved were:

- **NLP model** $U = 0.465680$
- **Case Study** $U = 0.881101$ (SoPC + cordic)

The objective in the case study was to choose one part of the application to move to hardware that resulted in a significant decrease in U. The automated partitioner however selected many nodes to move to hardware since this achieved the lowest U. However the task of converting the 28 selected functions into hardware would involve a large amount of design effort. To make the results applicable to real-world design problems, the effort to synthesize the partitioned nodes should be included in the partitioning solver. This could be done by using a hardware capacity vector, $\vec{Sz}[hw]$, as suggested in Section 5.2.5. $\vec{Sz}[hw]$ would have two elements: size and design time. This would require that each node also be annotated with an estimate of the time to convert the design to hardware. For those nodes with pre-existing hardware designs, the design time would be zero.

Another observation that can be made is related to the functional pairing of nodes. Some nodes are connected by common data structures: for example MinHeap::insert and MinHeap::pop. In this case they were assigned to the same implementation target (hardware). However they could have been placed assigned to different implementation targets. If the related nodes could be identified, the model could be refined by adding a constraint that such related nodes be assigned to the same implementation target.

7.5 Heuristic Gain Evaluation

In this section the results of an evaluation of the four proposed gain metrics are reported. At each step of the partitioning heuristic (Algorithm 6.1) the node with the largest gain is selected to be moved. Four gain metrics were proposed in Chapter 6. They were:

1. g^c – decrease in node worst-case execution time,
2. g^χ – decrease in scaled, weighted cutset,
3. g^U – decrease in processor utilization, and
4. g^U/sz – decrease in processor utilization divided by node hardware size.

Table 7.9: Gain Experiment Results Summary

	Run-Time (seconds)				# Feasible			
Problem	g^c	g^χ	g^U	g^U/sz	g^c	g^χ	g^U	g^U/sz
idle eng	0.14	0.10	0.12	0.26	0	0	31	100
p100_1	0.65	4.99	0.52	0.57	0	40	82	83
p100_2	0.41	0.32	0.35	0.37	0	3	100	100
p200_1	31.18	0.68	25.11	25.52	92	0	92	98
p200_2	0.69	0.63	1.57	1.84	0	3	100	100
p500_1	5.61	2.72	97.92	120.93	0	0	4	0
p500_2	3.92	2.26	2.45	3.14	0	5	100	99
p1000_1	41.16	129.31	204.37	207.23	0	14	68	52
p1000_2	13.56	6.17	7.12	9.03	0	1	100	100
				Total	92	66	677	732

These gain metrics were compared using the idle engine test case and the eight test cases of GPS1. The heuristic was run 100 times, per gain metric, per test case. The run-times and number of feasible solutions are reported in Table 7.9.

Examining the total number of feasible solutions generated by gain metric (Table 7.9), it can be seen that g^c and g^χ produced far fewer feasible solutions than did g^U and g^U/sz (7 to 11 times fewer). This result was as expected. g^c reduces task worst-case execution times, C_i, without considering task frequency. g^χ reduces the number of kernel invocations but ignores task node worst-case execution times. An unexpected result was that g^c and g^χ did not reduce run-times: in 5 of 9 cases, run-times for g^U and g^U/sz were shorter than for one of g^c and g^χ. Optimizing for processor utilization resulted in more feasible solutions than optimizing for cutset or node worst-case execution times, and had comparable run-times.

Table 7.10: Processor Utilization Results for g^U and g^U/sz

	g^U		g^U/sz	
Problem	U min	U std dev	U min	U std dev
idle eng	0.4657	0.01881	0.4657	0.01350
p100_1	0.8984	0	0.8984	0
p100_2	0.8512	0	0.8512	0
p200_1	0.6772	0.02347	0.6772	0.02316
p200_2	0.7836	0	0.7836	0
p500_1	0.5538	0.006246	—	—
p500_2	0.7064	0	0.7064	0
p1000_1	0.7739	0.04015	0.8275	0.02619
p1000_2	0.2445	0	0.2445	0

The run-time using g^U/sz doesn't exceed the run-time using g^U by more than 30% in all cases except the idle engine test case were the run-time was approximately double. The total number of feasible solutions is 8% higher for g^U/sz than g^U, although on a case by case basis neither was consistently better than the other. To further compare these two gain metrics, the processor utilization minimum and standard deviation are reported per gain metric, per test case in Table 7.10.

Examining Table 7.10 shows that the minimum processor utilizations obtained using g^U and g^U/sz were the same in seven of nine cases. In one of the other cases, p1000_1, g^U performed better, and in the other, p500_1, no feasible solution was obtained using g^U/sz. There was little difference in the standard deviations in processor utilization obtained using either gain metric. From the results of Tables 7.10 and 7.9, it appears that scaling g^U by node hardware size provides little or no benefit and has slightly longer run-times.

7.5.1 Partitioned Kernel and Task Comparison

The best partitions generated using g^U were examined for the idle engine test case and each of the test cases in GPS2. The fraction of kernel nodes assigned to hardware and the fraction of application task nodes assigned to hardware are recorded in Table 7.11. In all cases, some kernel nodes were assigned to hardware, whereas sometimes no task nodes were assigned to hardware. The fraction of kernel nodes assigned to hardware was higher than the fraction of task nodes assigned to hardware, in all cases except the idle engine test case. That the automated partitioner always assigned some kernel nodes to hardware indicates that including the kernel in the partitioning is beneficial: it gives the partitioner more choices in trying to satisfy schedule feasibility. This preference for kernel nodes is explained by examining their contribution to processor utilization. The kernel is invoked more often than any task, which helps to increase the gain of moving the node to hardware. Also, unlike task edges, cut kernel edges do not incur additional cost. This give the partitioner more freedom to put any kernel node into hardware.

7.5.2 Repair Algorithm Evaluation

The heuristic gain experiment results can also be used to evaluate the repair algorithm (Algorithm 6.5) which attempts to fix a partition that does not pass the EDF analysis. For all nine test cases and all four gain metrics, the repair algorithm was invoked 363 times and was successful 26 times, with a resulting success rate of 7.16%. (See Appendix Tables B.5 - B.13 for repair data by test case and by gain metric.) This demonstrates that the repair algorithm does work, although with a low success rate. Furthermore, it only had success repairing partitions generated using g^χ. The run-times are also higher for those solutions upon which repairs are attempted. Therefore, instead of attempting to

Table 7.11: Nodes Assigned to Hardware

Problem	Kernel	Task
idle eng	45.4%	76.7%
p100_1	36.4%	3.0%
p100_2	36.4%	1.0%
p200_1	9.1%	2.5%
p200_2	45.4%	0%
p500_1	45.4%	6.2%
p500_2	63.6%	0%
p1000_1	36.4%	3.3%
p1000_2	54.5%	0.2%

repair partitions, it may be more beneficial to spend the effort on longer run lengths of the heuristic.

7.6 Summary

The results of comparing the NLP model and heuristic partitioners were presented in this chapter. Each was used to partition concurrent applications and their kernels into hardware and software in order to pass the EDF feasibility analysis. The NLP model minimized processor utilization U. This was also found to be the best goal for the FM-based heuristic. In order to test the partitioners, two data sources were used: the idle engine application from the case study, and generic test case sets GPS1 and GPS2. In order to use the idle engine data, the node hardware size and worst-case execution time had to be estimated. The data from the cs2 coprocessor was of limited use since it didn't fit neatly into the function call graph that Slif uses. A representation at a lower level, for instance a CDFG, would have accommodated the cs2 coprocessor better. However adopting the Slif graph representation allowed use of the gpslifgen tool to produce generic test cases. The generic test cases required fabrication of scheduling parameters and hardware capacities.

The heuristic was shown to produce near-optimal results and to run three orders of magnitude faster than the NLP model solver. Evaluating the partitioning results for the idle engine test case revealed that the cost of synthesis should be included in the partitioner to make the results of practical use. The repair algorithm, while shown to function correctly, had a low success rate and increased run-times. Therefore it is concluded that the repair algorithm should not be used or a different approach investigated.

Chapter 8

Summary and Concluding Remarks

In this dissertation, the results of a study on the relationship between partitioning and scheduling of real-time embedded systems have been presented. In particular the relationship between hardware/software partitioning of uni-processor systems and scheduling by the preemptive Earliest Deadline First (EDF) was studied. The study was divided into three parts: EDF feasibility analysis for task sets with application coprocessors, a case study in hardware/software partitioning of an application and real-time kernel, and automated hardware/software partitioning of applications and kernels so as to pass EDF feasibility analysis.

8.1 EDF Feasibility Analysis

In the first study, the EDF feasibility analysis algorithm developed by Stankovic *et al* [91] was extended to task sets that include a task that blocks on coprocessors multiple times. The extended analysis accounts for the parallel execution which can occur between the processor and coprocessor.

8.1.1 Contributions

The extension of EDF feasibility analysis for coprocessors made the following contributions.

- Identified and characterized the problem of accounting for application coprocessors in EDF feasibility analysis.
- Proposed a first solution for the EDF feasibility analysis for task sets that include coprocessor use.
- Extends the study of an important real-time scheduling policy to the codesign process. Many embedded systems designed by the hardware/software codesign process are real-time systems. This work helps to make the EDF scheduling policy a viable option for hardware/software codesign.

8.1.2 Future Research

The work in EDF feasibility analysis for task sets with coprocessors can be extended and improved upon in four ways as described below.

- The analysis accounts for the parallel execution between processor and coprocessor. However it imposes limitations on tasks that have short relative deadlines and little slack. This limitation was made evident when the SoPC with cordic coprocessor was analyzed for schedule feasibility (Section 4.5). An improvement to this work would be to develop an algorithm that accounts for the parallel execution without imposing this limitation; perhaps by shifting slack between subtasks as suggested in Section 3.2.4.
- Extend the analysis for task sets that include multiple tasks that block on (multiple) coprocessors.

- Extend the analysis to multi-processor systems.
- Develop dynamic planning algorithms for EDF that incorporate tasks employing coprocessors.

This first study analyzed a specific way in which partitioning an application into software with hardware coprocessors affects scheduling.

8.2 Case Study

A case study of a System on Programmable Chip (SoPC) was presented. The target was a development board with Field Programmable Gate Array (FPGA) that had a soft-core processor. A real-time kernel was developed that provided the following services: task scheduling by the EDF policy, inter-task message queues, and application coprocessor integrations. An application was developed that used the kernel services to schedule periodic and aperiodic tasks. The application missed deadlines when implemented completely in software, so manual partitioning of the application and kernel was performed.

8.2.1 Contributions

The contributions to study of embedded systems made by this case study are listed below.

- This case study added to the relatively small body of research on hardware/software partitioning of the kernel. (The other studies reviewed in Chapter 2 were the δ Framework and the Spring scheduling coprocessor - SSCoP).
- A metric (ΔU/LE) was introduced for the comparison of kernel and application processors. It measures the improvement to processor utilization per unit of hardware

used (ΔU/LE). Without scheduling information, it is difficult to assess the benefit that a coprocessor offers the application. The metric can be used for both kernel and application coprocessors.

- A real-time kernel was developed that had integrated support for application coprocessors.
- The case study also provides data which is measured, not estimated, which can be used to help evaluate automated hardware/software partitioners.

8.2.2 Future Research

The case study has potential for future research in three directions.

- The case study was not entirely authentic because the idle engine and environmental input were simulated on the same chip as the controllers. Additional case studies of real-time SoC/SoPC applications would generate better data on which to test partitioning and scheduling algorithms.
- Implement the nodes of the kernel Slif graph on the FPGA. This would have two benefits: it would verify the hardware size and time estimates used in Chapter 7, and it could be used to confirm the automated partitioner results with regard to schedule feasibility.
- The case study focused on an SoPC with one processor to which coprocessors were added to enhance schedule feasibility. A question that arises is this, "Which is more beneficial: adding processors or coprocessors (application specific hardware)?" Kernel overhead may be higher for multi-processor systems than for single-processor

systems. However, the added processor could be considered as a more flexible co-processor, available for use by a wide range of tasks, yet with slower execution time than a custom hardware coprocessor. This question adds another dimension to the hardware/software partitioning problem.

The case study demonstrated application partitioning and kernel partitioning, and comparison of their results.

8.3 Hardware/Software Partitioning

A definition of the hardware/software partitioning problem was declared for single-processor systems. The goal was to partition systems such that all deadlines were met when scheduled by the preemptive EDF policy. In addition, kernel partitioning was incorporated with application partitioning. The problem definition was explored with a non-linear programming (NLP) model and a heuristic based upon the Fiduccia/Mattheyses circuit partitioning heuristic.

8.3.1 Contributions

Development of the automated partitioners and their evaluation provided some insights and contributions to the study of hardware/software codesign.

- The evaluation of real-time schedule feasibility for concurrent systems was integrated with hardware/software partitioning. In particular EDF schedule analysis was integrated into the partitioning problem.

- Results indicate that minimization of processor utilization U is a better partitioning objective than minimizing node worst-case execution times or minimizing cutsets.

- Partitioning of kernels was joined with partitioning of applications. The results showed that kernel nodes were often selected for implementation in hardware, indicating that including the kernel is a positive contribution to the partitioning process.

- Design time was identified as an important metric in hardware/software partitioning. Comparing results from the automated partitioner with the case study showed that design cost accounting should also be included in the automated partitioners. This would probably make kernel partitioning even more desirable because once it is partitioned, it can be reused for multiple applications with little design effort. The same applies for commonly used application functions: for example, data structure or mathematical functions.

- The partitioning results support the thesis statement that partitioning and scheduling of concurrent systems are best approached in an integrated manner. When the heuristic partitioner did not directly minimize processor utilization, relatively few partitions with feasible schedules were produced (Section 7.5). In other words, the scheduling problem is hard enough that it is not likely to be satisfied unless directly addressed by the partitioner.

8.3.2 Future Research

The work on automated hardware/software partitioning can be extended in several ways.

- The Slif graph representation was adopted for this work, in part due to the availability of the automated generic test case generator (gpslifgen). However, as discussed in Section 7.1, representing an application at the function level may limit the partitioner's choices. Representation at a lower level, such as the control and data-flow graph may yield better partitioning results. At the same time it would probably increase the complexity of the partitioning model due to the presence of conditional branches.

- The partitioning problem, as defined, had two implementation targets: a hardware target and a software target. The partitioners could be extended by accommodating multiple implementation targets. As suggested in Section 8.2.2, it would be interesting to compare the trade-offs between addition of coprocessors and use of multiple software targets.

- Another direction in which this work could be taken is to explore hardware synthesis from software descriptions (i.e. a form of behavioral synthesis). The automated partitioning approach assumes that hardware and software implementations are available for each node. This is not practical due to the effort required to design each node in hardware and software.

  One approach to this problem is specification in SystemC [41]. SystemC adds hardware constructs to the C++ software implementation language. These hardware constructs facilitate generation of code for hardware synthesis or can be left entirely in software and compiled. This does however impose a style on the designer that mixes hardware and software design constructs.

  If software implementation code could be automatically transformed into code for hardware synthesis, it would probably simplify the task of the designer. A new approach being investigated by Hounsell *et al* [49] is called "coprocessor synthesis". They start by identifying software functions for implementation in hardware. The object code of the compiled function is used to generate a coprocessor which is actually a small micro-programmed processor that is optimized for that function. One possible drawback to this approach is that speed-up of the function might be limited because the function is essentially being executed in software form.

Hardware and software designs that perform the same task can be based on different paradigms. For example the software min-heaps of the case study kernel were transformed into round-robin bidding in the cs2 coprocessor. Such non-linear transformations could be very complex for a synthesis tool to perform, requiring innovative techniques.

The hardware/software partitioning study revealed the importance of integrating schedule feasibility, and the benefit of including the kernel.

8.4 Final Observations

The inter-related nature of hardware/software partitioning, and scheduling of concurrent systems has been demonstrated through three studies. The importance of integrating schedule feasibility into the partitioning stage has been demonstrated. In the process, scheduling by the preemptive Earliest Deadline First policy has been explored in its relation to hardware/software codesign.

Bibliography

[1] Jay K. Adams and Donald E. Thomas. Multiple-process behavioural synthesis for mixed hardware-software systems. In *International Symposium on System Synthesis*, pages 10–15, September 1995.

[2] Joakim Adomat, Johan Furunäs, Lennart Lindh, and Johan Stärner. Real-time kernel in hardware rtu: A step towards deterministic and high-performance real-time systems. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 164–168, 1996.

[3] Altera Corporation. *Nios Embedded Processor: 32-Bit Programmers Reference Manual*, 2002.

[4] Altera Corporation. *Nios Embedded Processor Development Board Data Sheet*, 2003.

[5] Shawki Areibi and Anthony Vannelli. *Advanced Search Techniques for Circuit Partitioning*, pages 77–98. American Mathematical Society, 1994.

[6] ASICS World Services, LTD. *Enhanced Floating Point Unit IP Core*, 2003.

[7] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems Journal 3*, 1991.

[8] Felice Balarin, Massimiliano Chiodo, Attila Jurecska, Luciano Lavagno, Bassam Tabbara, and Alberto Sangiovanni-Vincentelli. Automatic generation of a real-time operating system for embedded systems: Extended abstract. In *International Workshop on Hardware/Software Co-Design (CODES/CACHE)*, Braunschweig, Germany, 1997.

[9] Felice Balarin, Luciano Lavagno, Praveen Murthy, and Alberto Sangiovanni-Vincentelli. Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, pages 71–82, 1998.

[10] A. Balboni, W. Fornaciari, and D. Sciuto. Partitioning and exploration strategies in the tosca co-design flow. In *International Workshop on Hardware/Software Co-Design (CODES/CACHE)*, pages 62–69, 1996.

[11] Andrea Balluchi, Luca Benvenuti, Maria D. Di Benedetto, Tiziano Villa, Howard Wong-Toi, and Alberto L. Sangiovanni-Vincentelli. Hybrid controller synthesis for idle speed management of an automotive engine. In *Proceedings of the American Control Conference*, pages 1181–1185, Chicago, Illinois, June 2000.

[12] Earl R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM Journal on Algebraic and Discrete Methods*, 3(4):541–550, December 1982.

[13] Peter Barth. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Germany, 1995.

[14] S. K. Baruah, L. E. Rosier, and R. R. Howell. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of IEEE Real-Time Systems Symposium*, 1990.

[15] Peter Bellows and Brad Hutchings. Jhdl-an hdl for reconfigurable systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 175–184, 1998.

[16] T. Benner and R. Ernst. An approach to mixed systems co-synthesis. In *Proceedings, 5th International Workshop on Hardware/Software Codesign*, pages 9–14. IEEE Computer Society Press, 1997.

[17] Thomas Benner and Rolf Ernst. A combined partitioning and scheduling algorithm for heterogeneous multiprocessor systems. Research Report CY–96–2, Institute of Computer Engineering, Technical University of Braunschweig, 1996.

[18] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[19] Gérard Berry. *The Esterel v5 Language Primer*. Centre de Mathématiques Appliquées, Sophia-Antipolis, France, 1999.

[20] Ivo Bolsens, Hugo J. De Man, Bill Lin, Karl van Rompaey, Steven Vercauteren, and Diederik Verkest. Hardware/software co-design of digital telecommunication systems. *Proceedings of the IEEE*, 85(3):391–418, March 1997.

[21] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, 4:155–182, April 1994.

[22] Wayne Burleson, Jason Ko, Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles Weems. The spring scheduling coprocessor: A scheduling accelerator. *IEEE Transactions On VLSI Systems*, 7:38–47, 1999.

[23] Karam S. Chatha and Ranga Vemuri. Magellan: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *CODES*, pages 42–47, 2001.

[24] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems Journal 2*, 1990.

[25] Massimiliano Chiodo, Paolo Giusto, Attila Jurecska, Harry C. Hsieh, Alberto Sangiovanni-Vincentelli, and Luciano Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, pages 26–36, August 1994.

[26] Pai Chou, Elizabeth A. Walkup, and Gaetano Borriello. Scheduling for reactive real-time systems. *IEEE Micro*, 14(4):37–47, August 1994.

[27] E. F. Codd. Multiprogram scheduling. *Communications of the ACM*, 3(6, 7):347–350,413–418, 1960.

[28] Marco Cornero, Filip Thoen, Gert Goossens, and Franco Curatelli. *Software Synthesis for Real-Time Information Processing Systems*, pages 260–279. Kluwer Academic Publishers, 1995.

[29] Rolf Ernst. Codesign of embedded systems: Status and trends. *IEEE Design & Test of Computers*, pages 45–54, April-June 1998.

[30] Rolf Ernst, Jörg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[31] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation Conference*, pages 175–181, Las Vegas, Nevada, 1982.

[32] Franz Fischer, Annette Muth, and Georg Färber. Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment. In *Sixth International Workshop on Hardware/Software Co-Design*, Seattle, Washington, 1998. IEEE.

[33] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* PTR Prentice Hall, 1994.

[34] David Galloway. The transmogrifier c hardware description language and compiler for fpgas. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[35] M. Garey and D. Johnson. *Computers and Intractability.* Freeman, 1979.

[36] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Research Report RR-2966, INRIA, Le Chesnay, France, 1996.

[37] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[38] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[39] Rajesh K. Gupta, Claudionor N. Coelho Jr., and Giovanni De Micheli. Program implementation schemes for hardware-software systems. *IEEE Computer*, pages 48–55, January 1994.

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

[41] Ali Habibi and Sofiène Tahar. A survey of system-on-a-chip design languages. In *Proceedings of the Third International Workshop on System-on-Chip for Real-Time Aplications*, 2003.

[42] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language lustre. another look at real time programming. *Proceedings of the IEEE*, September 1991.

[43] D. Havel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[44] J. Henkel, Th. Benner, R. Ernest, W. Ye, N. Serafimov, and G. Glawe. Cosyma: A software-oriented approach to hardware/software co-design. *The Journal of Computer and Software Engineering*, 2(3):293–314, 1994.

[45] Jörg Henkel and Rolf Ernst. The interplay of run-time estimation and granularity in hw/sw partitioning. In *International Workshop on Hardware/Software Co-Design (CODES/CACHE)*, pages 52–58, 1996.

[46] Jörg Henkel and Yanbing Li. Energy-conscious hw/sw-partitioning of embedded systems: A case study on an mpeg-2 encoder. In *Sixth International Workshop on Hardware/Software Co-Design*, Seattle, Washington, 1998. IEEE.

[47] Thomas Hollstein, Jürgen Becker, Andreas Kirschbaum, and Manfred Glesner. Hipart: A new hierarchical semi-interactive hw/sw partitioning approach with fast debugging for real-time embedded systems. In *Sixth International Workshop on Hardware/Software Co-Design*, Seattle, Washington, 1998. IEEE.

[48] Denis Hommais, Frédéric Pétrot, and Ivan Augé. A practical tool box for system level communication synthesis. In *CODES*, pages 48–53, 2001.

[49] Ben Hounsell and Richard Taylor. Co-processor synthesis: A new methodology for embedded software acceleration. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[50] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[51] A. A. Jerraya, M. Romdhani, C. A. Valderrama, Ph. Le Marrec, F. Hessel, G. F. Marchioro, and J. M. Daveau. *Languages for System-Level Specification and Design*, chapter 7. Hardware/Software Codesign: Principles and Practice. Kluwer Academic Publishers, Netherlands, 1997.

[52] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrainted hardware/software partitioning problem. In *CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, pages 42–48, 1994.

[53] Asawaree Kalavade and Edward A. Lee. The extended partitioning problem: Hardware/software mapping and implementation-bin selection. In *Sixth International Workshop on Rapid Systems Prototyping*, 1995.

[54] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[55] Kurt Keutzer. Hardware-software co-design and esda. In *Design Automation Conference*, pages 435–436, San Diego, California, 1994.

[56] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[57] Michael J. Knieser and Christos A. Papachristou. Comet: A hardware-software codesign methodology. In *Proceedings European Design Automation Conference*, pages 178–183, Geneva, Switzerland, 1996.

[58] B. Krishnamurthy. An improved min-cut algorithm for partitioning vlsi networks. *IEEE Transactions on Computers*, C-33(5):438–446, May 1984.

[59] David C. Ku and Giovanni De Micheli. Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):696–718, June 1992.

[60] Edward A. Lee. Overview of the ptolemy project. UCB/ERL Memorandum M98/72, Department of Electrical Engineering and Computer Science, University of California, Berkeley, February 1999.

[61] Edward A. Lee. What's ahead for embedded software? *IEEE Computer*, pages 18–26, September 2000.

[62] Jaehwan Lee, Vincent John Mooney III, Anders Daleby, Karl Ingström, Tommy Klevin, and Lennart Lindh. A comparison of the rtu hardware rtos with a hardware/software rtos. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 683–688, January 2003.

[63] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, London, 1990.

[64] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architecture. In *CODES*, pages 507–512, 2000.

[65] Clifford Liem, François Naçabal, Carlos Valderrama, Pierre Paulin, and Ahmed Jerraya. System-on-a-chip cosimulation and compilation. *IEEE Design & Test of Computers*, pages 16–25, April-June 1997.

[66] Lennart Lindh and Frank Stanischewski. Fastchart-idea and implementation [virtual machine]. In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 401–404, 1991.

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

[68] M. L. López-Vallejo, J. Grajal, and J. C. López. Constraint-driven system partitioning. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, Paris, France, 2000.

[69] Jan Madsen, Jesper Grode, and Peter V. Knudsen. *Hardware/Software Partitioning using the LYCOS System*, chapter 9. Hardware/Software Codesign: Principles and Practice. Kluwer Academic Publishers, Netherlands, 1997.

[70] Hugo J. De Man, Ivo Bolsens, Bill Lin, Karl van Rompaey, Steven Vercauteren, and Diederik Verkest. *Co-Design of DSP Systems*, pages 75–104. Kluwer Academic Publishers, Netherlands, 1996.

[71] Giovanni De Micheli and Rajesh K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, March 1997.

[72] Vincent J. Mooney III and Douglas M. Blough. A hardware-software real-time operating system framework for socs. *IEEE Design & Test of Computers*, 2002.

[73] Vincent J. Mooney III, Claudionor N. Ceolho Jr., Toshiyuki Sakamoto, and Giovanni De Micheli. Synthesis from mixed specifications. In *Proceedings European Design Automation Conference*, pages 114–119, 1996.

[74] Andrew Morton. Partitioning hardware/software codesigns for embedded systems. Phd research proposal, University of Waterloo, 2000.

[75] Andrew Morton and Wayne M. Loucks. Real-time kernel support for coprocessors: Empirical study of an sopc. In *Proceedings of the Embedded Systems and Applications Conference*, pages 10–15, 2003.

[76] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 869–875, 2004.

[77] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *Proceedings of the 12th TRON Project International Symposium*, pages 34–42, 1995.

[78] Ralf Niemann and Peter Marwedel. *An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming*, chapter 1. Design Automation for Embedded Systems. Kluwer Academic Press, 1997.

[79] Richard O'Donnell. Prolog to hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):965–966, July 1994.

[80] OpenBSD. src/lib/libm/src/k_cos.c - view - 1.2. World wide web document, http://www.openbsd.org, 2002.

[81] Achim Österling, Thomas Benner, Rolf Ernst, Dirk Herrmann, Thomas Scholz, and Wei Ye. *The Cosyma System*, chapter 8. Hardware/Software Codesign: Principles and Practice. Kluwer Academic Publishers, Netherlands, 1997.

[82] JoAnn M. Paul, Simon N. Peffers, and Donald E. Thomas. Frequency interleaving as a codesign scheduling paradigm. In *CODES*, pages 131–135, 2000.

[83] José L. Pino, Michael C. Williamson, and Edward A. Lee. Interface synthesis in heterogeneous system-level dsp design tools. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Atlanta, Georgia, May 1996.

[84] RealFast. Ultrafast micro kernel (ufμk) - hw os accelerator. World wide web document, http://www.realfast.se/rfipp/products/s16/UFK_datasheet.pdf, 2002.

[85] David L. Rhodes and Wayne Wolf. Overhead effects in real-time preemptive schedules. In *CODES*, pages 193–197, 1999.

[86] Ismail Ripoll, Alfons Crespo, and Aloysius K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, July 1996.

[87] D. Saha, R. S. Mitra, and Anupam Basu. Hardware software partitioning using genetic algorithm. In *Proceedings of 10th International Conference on VLSI Design*, pages 155–160, 1996.

[88] L. A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[89] M. Schwiegershausen, H. Kropp, and P. Pirsch. A system level hw/sw partitioning and optimization tool. In *Proceedings European Design Automation Conference*, pages 120–125, 1996.

[90] Neil L. Shipp. Co-simulation of combined hardware software systems. Master's thesis, University of Waterloo, 1996.

[91] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[92] M. B. Strivastava and R. W. Broderson. Rapid-prototyping of hardware and software in a unified framework. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 152–155, 1991.

[93] Filip Thoen, Marco Cornero, Gert Goossens, and Hugo De Man. Real-time multitasking in software synthesis for information processing systems. In *Eighth International Symposium on System Synthesis*, pages 48–53, California, 1995. IEEE Computer Society Press.

[94] University of California. *POLIS: A Design Environment for Control-Dominated Embedded Systems, version 0.3 User's Manual*, 1997.

[95] Frank Vahid. Modifying min-cut for hardware and software functional partitioning. In *International Workshop on Hardware/Software Co-Design (CODES/CASHE)*, pages 43–48, Braunschweig, Germany, 1997.

[96] Frank Vahid. Techniques for minimizing and balancing i/o during functional partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):69–75, January 1999.

[97] Frank Vahid and Thuy dm Le. Towards a model for hardware and software functional partitioning. In *International Workshop on Hardware/Software Co-Design (CODES/CASHE)*, pages 116–123, Pittsburgh, Pennsylvania, 1996.

[98] Frank Vahid and Daniel D. Gajski. Specification partitioning for system design. In *Proceedings of 29th ACM/IEEE Design Automation Conference*, pages 219–224, 1992.

[99] C. A. Valderrama, M. Romdhani, J. M. Daveau, G. Marchioro, A. Changuel, and A. A. Jerraya. *Cosmos: A Transformational Co-Design Tool for Multiprocessor Architectures*, chapter 10. Hardware/Software Codesign: Principles and Practice. Kluwer Academic Publishers, Netherlands, 1997.

[100] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Hardware/software partitioning of embedded system in ocapi-xl. In *CODES*, pages 30–35, 2001.

[101] Steven Vercauteren, Bill Lin, and Hugo J. De Man. Constructing application-specific heterogeneous embedded architectures for custom hw/sw applications. In *Proceedings of 33rd ACM/IEEE Design Automation Conference, DAC-96*, pages 521–526, Las Vegas, NV, June 1996.

[102] J. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959.

[103] J. Walther. A unified algorithm for elementary functions. In *Spring Joint Computer Conference*, pages 379–385, 1971.

[104] Michael Wang, Katsuharu Suzuki, and Wayne Dai. Memory and logic integration for system-in-a-package. In *Proceedings of the 4th International Conference on ASIC*, October 2001.

[105] Wind River Systems, Inc. *VxWorks Programmer's Guide 5.3.1*, 1 edition, 1998.

[106] Wayne Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[107] Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rundy Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Design & Test of Computers*, 18(5):46–58, 2001.

[108] Q. Zheng and K. G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, 42(2), 1994.

Appendix A

EDF Scheduling

A.1 Upper Bound Definition

Upper bounds are set on the interval over which the processor demand analysis is performed. Derivation of two of these bounds, ub_1 and ub_2, is explained here. The explanation is adapted from [91] where the following theorem is employed:

Theorem A.1 (Baruah et. al. [14]). *If τ is not feasible and $U < 1$, then $h(t) > t$ implies $t < D_{max}$ or $t < \max_{i=1,\ldots,n}\{T_i - D_i\}\frac{U}{1-U}$.*

Proof. Assume $h(t) > t$ and $t \geq D_{\max}$. It must be proved that $t < \max_{i=1,\ldots,n}\{T_i - D_i\}\frac{U}{1-U}$.
Given: $t < h(t)$.
By definition:

$$\begin{aligned}
h(t) &= \sum_{D_i \leq t}\left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i \\
&\quad \text{eliminate floor function and rearrange :} \\
&\leq \sum_{i=1}^{n}\left(\frac{t + T_i - D_i}{T_i}\right) C_i \\
&\leq t\sum_{i=1}^{n}\frac{C_i}{T_i} + \sum_{i=1}^{n}\frac{C_i}{T_i}(T_i - D_i) \\
&\leq tU + \max_{i=1,\ldots,n}\{T_i - D_i\}\sum_{i=1}^{n}\frac{C_i}{T_i} \\
h(t) &\leq tU + \max_{i=1,\ldots,n}\{T_i - D_i\}U
\end{aligned}$$

Substituting into $t < h(t)$:

$$\begin{aligned} t &< tU + \max_{i=1,\dots,n}\{T_i - D_i\}U \\ & \quad \text{rearrange and factor out } t: \\ t(1-U) &< \max_{i=1,\dots,n}\{T_i - D_i\}U \\ t &< \max_{i=1,\dots,n}\{T_i - D_i\}\frac{U}{1-U} \end{aligned}$$

□

Zheng and Shin [108] define an upper bound that can be found by substituting

$$h(t) \le \sum_{i=1}^{n}\left(\frac{t+T_i-D_i}{T_i}\right)C_i$$

from above into $t < h(t)$ to get:

$$\begin{aligned} t &< \sum_{i=1}^{n}\left(\frac{t+T_i-D_i}{T_i}\right)C_i \\ t &< t\sum_{i=1}^{n}\frac{C_i}{T_i} + \sum_{i=1}^{n}\left(\frac{T_i-D_i}{T_i}\right)C_i \\ t(1-U) &< \sum_{i=1}^{n}\left(\frac{T_i-D_i}{T_i}\right)C_i \\ t &< \frac{\sum_{i=1}^{n}\left(1-\frac{D_i}{T_i}\right)C_i}{1-U} \end{aligned}$$

Zheng and Shin's upper bound is:

$$ub_1 = \max\left\{D_{\max}, \frac{\sum_{i=1}^{n}\left(1-\frac{D_i}{T_i}\right)C_i}{1-U}\right\}.$$

George *et al* [36] similarly obtain another upper bound:

$$ub_2 = \frac{\sum_{D_i \le T_i}\left(1-\frac{D_i}{T_i}\right)C_i}{1-U}.$$

Appendix B

Partitioning Problem Data

B.1 Slif Pre-Processing

The partitioners of Chapters 5 and 6 use Slif graphs to describe the structure of tasks and the kernel. The Slif graph is a function access graph, meaning that each node represents a function and each (directed) edge represents a function invocation, where the edge source is the caller and the edge destination is the called function. The nodes are annotated with function execution-times for hardware and software (see Figures B.3 – B.9 for example Slifs). These are times for each invocation of the node (function) and are referred to here as base execution times. The edges are labeled with invocation counts which represent the number of times that the source node invokes the destination node per execution of the source node. These edge counts are referred to as base counts. Before the partitioners use the Slif graphs, the base execution times and base edge counts need to be converted to total execution times ($c_{i,j}[hw]$, $c_{i,j}[sw]$) and total edge counts ($f_{i,p}$).

Procedures `PropagateTree` and `PropagateEdge` are used to calculate total node execution times and total edge counts for a full traversal of the Slif graph. Procedure `PropagateTree` identifies the root node of Slif$_i$ (which has no incoming nodes). Its invocation count is defined to be one. The total execution time of the root node is therefore the same as the base execution time. Procedure `PropagateEdge` is invoked for each of the root node's outgoing edges.

Procedure `PropagateEdge` is a recursive procedure. It calculates the total edge count by multiplying the edge's base count by the invocation count of the calling node. The edge's destination node is then examined. If all of the node's incoming edges' total counts have been calculated, then their sum is assigned to the node invocation count. This is used to calculate the node's total execution time. The procedure recurses for each of the node's

```
find root node $n_{i,j}$ ;
$f_{i,j} \leftarrow 1$ ;
$c_{i,j}^{\text{total}} \leftarrow c_{i,j}^{\text{base}}$ ;
**foreach** *outgoing edge* $e_{i,p}$ *of* $n_{i,j}$ **do**
    call PropagateEdge($e_{i,p}, f_{i,j}$) ;
**end**
```

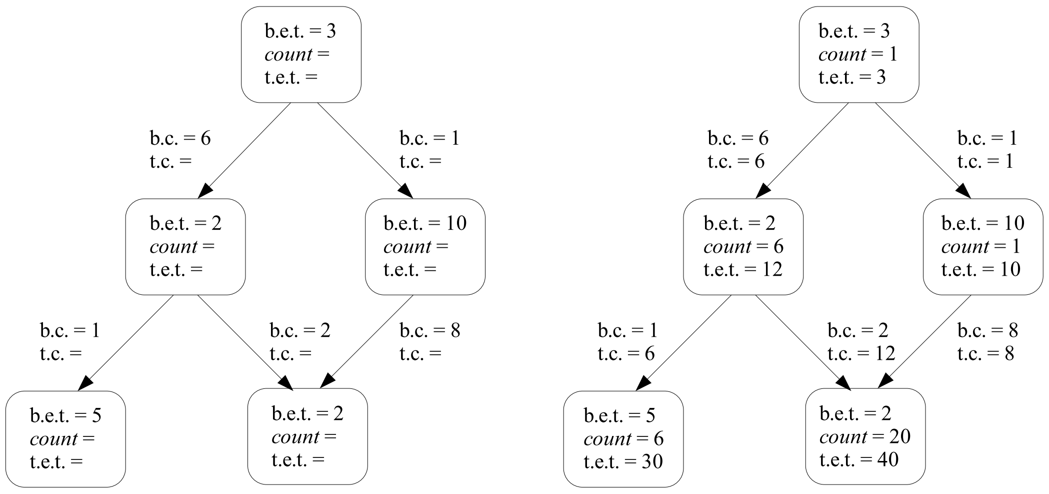
Procedure: `PropagateTree(`$Slif_i$`)`

outgoing edges.

```
$f_{i,p}^{\text{total}} \leftarrow f_{i,p}^{\text{base}} * f_{\text{in}}$ ;
identify destination node $n_{i,j}$ ;
**if** *all incoming edges* $e_{i,q}$ *of* $n_{i,j}$ *have* $f_{i,q}^{total}$ **then**
    $f_{i,j} \leftarrow \sum_{e_{i,q} \in \text{incoming edges}} f_{i,q}^{\text{total}}$ ;
    $c_{i,j}^{\text{total}} \leftarrow c_{i,j}^{\text{base}} * f_{i,j}$ ;
    **foreach** *outgoing edge* $e_{i,q}$ *of* $n_{i,j}$ **do**
        call PropagateEdge($e_{i,q}, f_{i,j}$) ;
    **end**
**end**
```

Procedure: `PropagateEdge(`$e_{i,p}, f_{in}$`)`

A small example is shown in Figure B.1. In Figure B.1(a) five nodes are shown with base execution time (b. e. t.) only and 5 edges with base count (b. c.) only. Figure B.1(b) shows, the same example after invoking Procedure `PropagateTree`: total execution times (t. e. t.) and total edge counts (t. c.) have been calculated.



(a) before (b) after

Figure B.1: `PropagateTree` Example

B.2 Idle Engine Data

B.2.1 Node Hardware Size Estimation

Node hardware worst-case execution time has been chosen as the most logical value from which to estimate node hardware size. However it is difficult to find literature that relates hardware size to hardware execution time. Out of necessity, however, a relationship needs to be chosen. It is proposed here that as hardware size (i.e. logic gates) increases, execution time also increases. In favour of this argument, consider that as the hardware size increases, the depth of combinational logic gates is likely to increase, increasing signal propagation delay. This argument is further supported by an example from industry. Asics World Services, Ltd [6] offers a suite of floating-point arithmetic blocks from which a floating-point unit can be built. There are four 32-bit blocks: add/subtract, multiply, divide and compare. There are also four 64-bit blocks implementing the same operations. Gate count and propagation delay are reported for a 0.18u process (the results for the 64-bit divide block are not provided). The data is plotted in Figure B.2. Linear regressions for the 32-bit and 64-bit blocks have correlation coefficients $r^2 = 0.96$ and $r^2 = 0.87$.

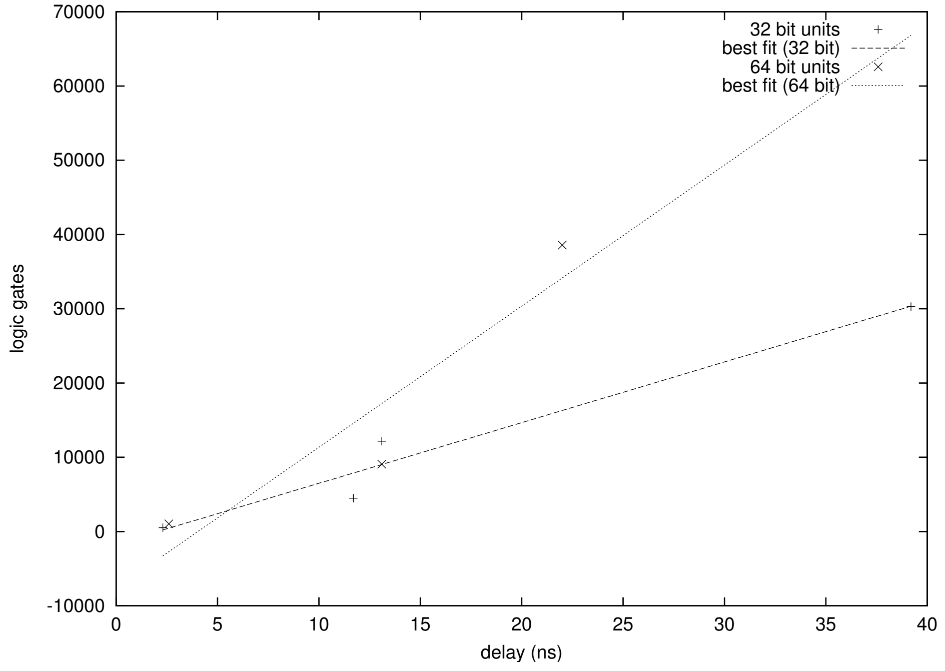


Figure B.2: ASICS.ws FPU Size vs Time

The linear relationship between hardware size and time (latency) is supported by the correlation coefficients for the floating-point block example. However that is not the complete picture. For example, a synchronous circuit can be made smaller and execution time increased by decreasing parallelism. Although the evidence of a linear relationship between hardware size and execution time is weak, out of necessity it will be assumed for the purposes of estimating node hardware sizes for the idle engine test case. The ratio of hardware size to hardware execution time, $R^{sz} = \frac{sz[hw]}{c[hw]}$, is derived from the two case study coprocessors:

1. Cordic coproc: R^{sz} = 3840 LE / 8.10x10^{-5}s = 4.74x10^{7} LE/s
2. cs2 coproc: R^{sz} = 2496 LE / 4.56x10^{-6}s = 5.47x10^{8} LE/s

B.2.2 Application and Kernel Slif Graphs

This section shows the kernel and task Slif graphs for the idle engine problem as used by the partitioners. Worst-case execution time ($c[sw]$ and $c[hw]$) is reported in cycles (of the system clock) and hardware size ($sz[hw]$) is reported in logic elements (LE). C++ library function names are distinguished using italics. Those functions that are bound to a target are marked with a *hw* or *sw* in the lower right corner of the box (see for example the context switch functions (_ldProcCtxt, _stProcCtxt) of Figure B.3).

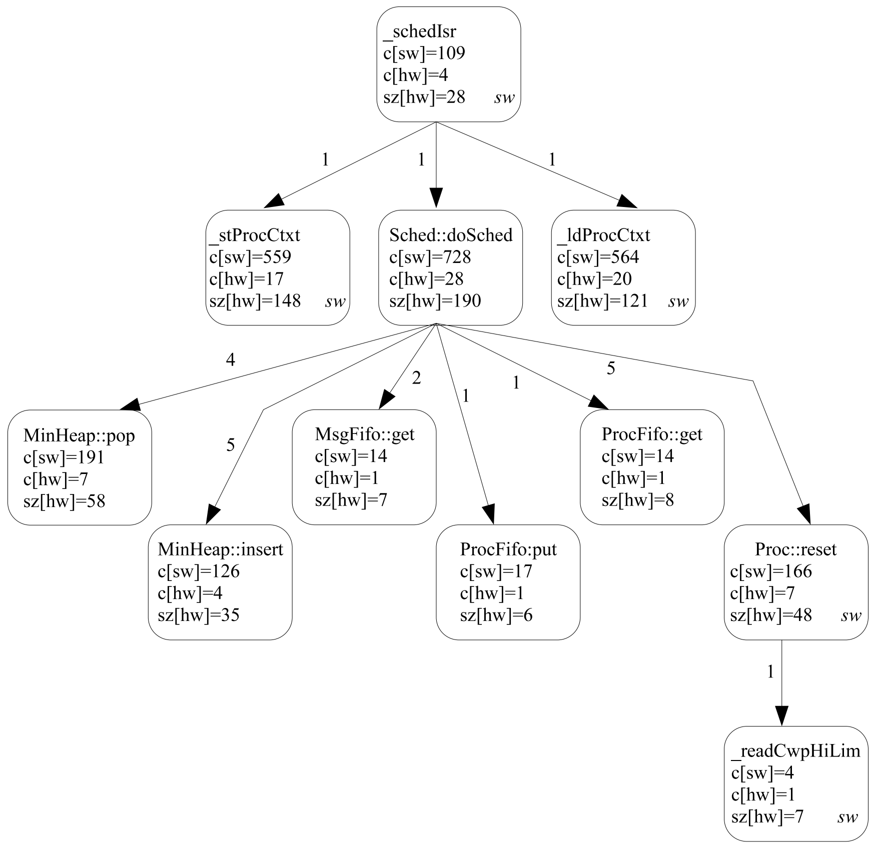


Figure B.3: Kernel Slif Graph

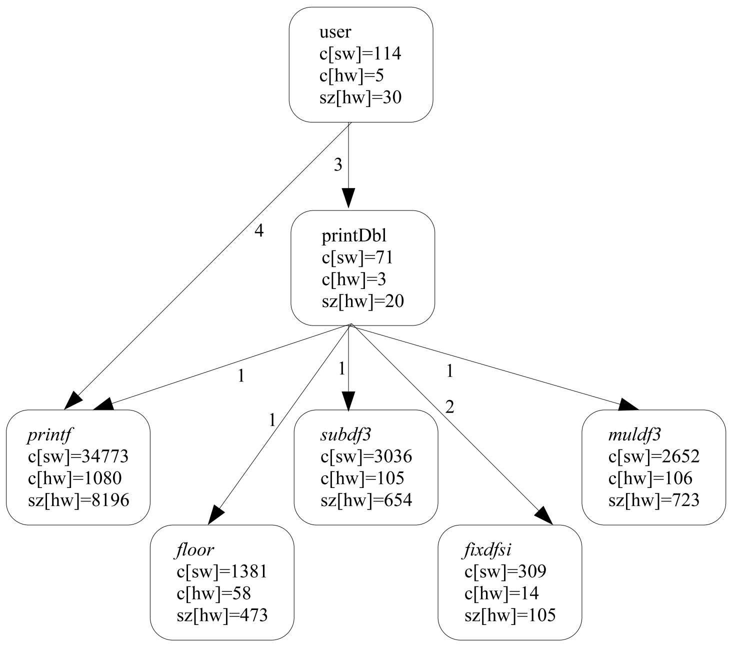


Figure B.4: User Interface Task Slif Graph

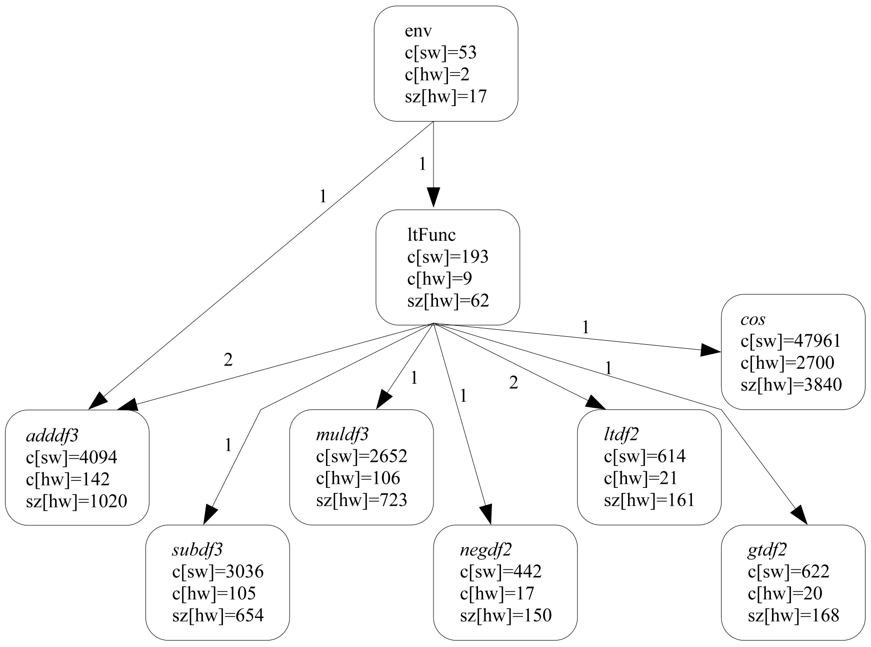


Figure B.5: Environment Task Slif Graph

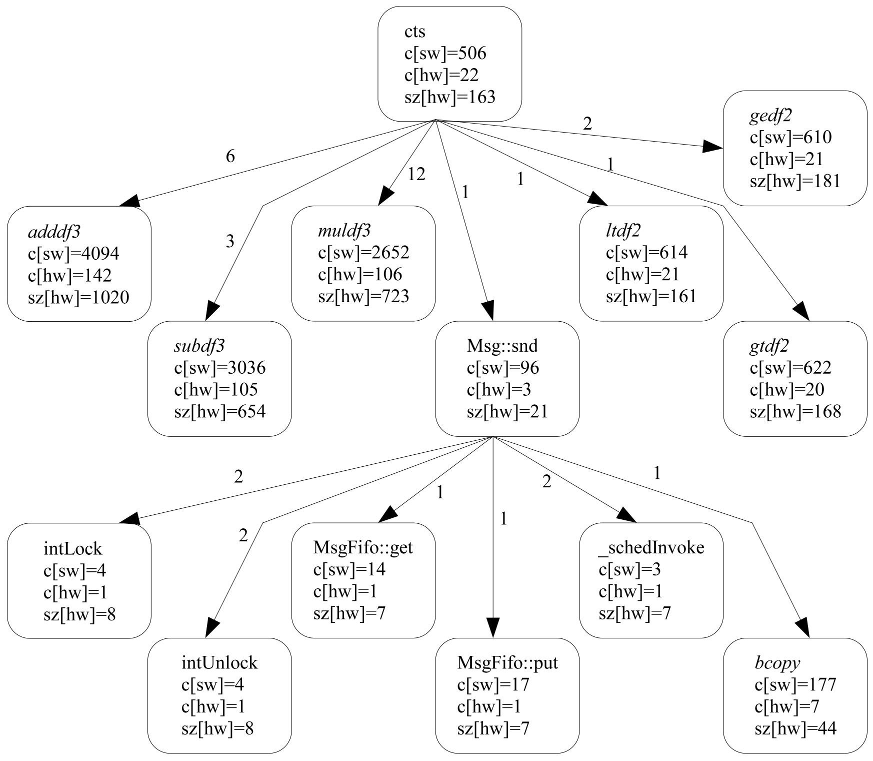


Figure B.6: CTS Task Slif Graph

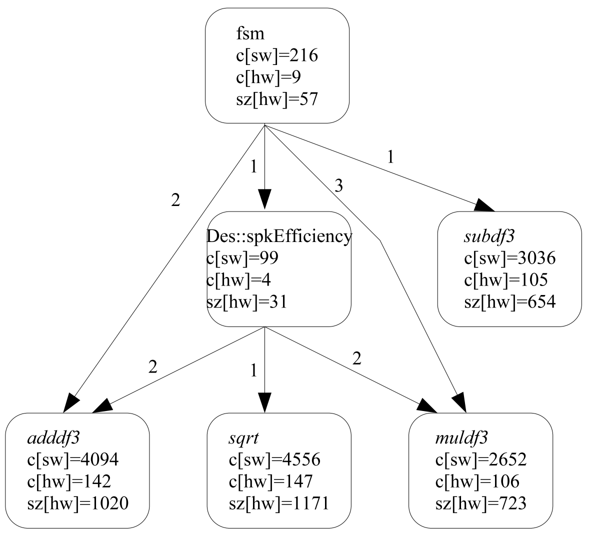


Figure B.7: FSM Task Slif Graph

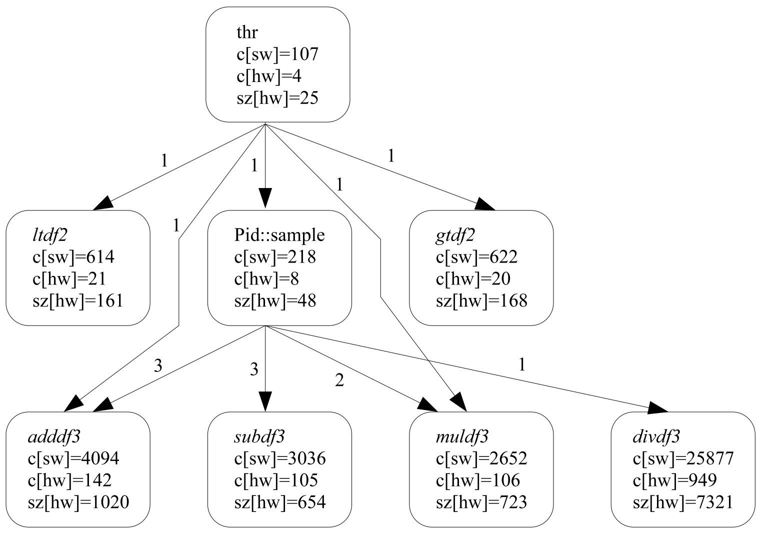


Figure B.8: Throttle Control Task Slif Graph

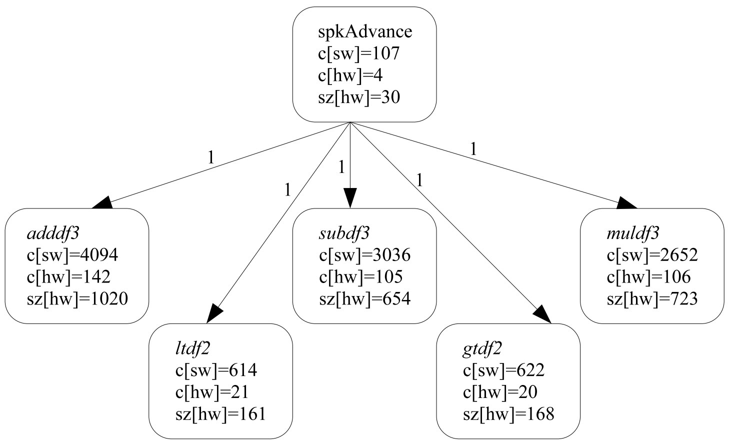


Figure B.9: Spark Advance Control Task Slif Graph

B.2.3 Partitioning Result

The idle engine problem, as represented in the preceding Slifs, was partitioned using the NLP model and the heuristic. The optimal result had processor utilization U = 0.465680 and used 4877 of the available 4914 LEs. The majority of nodes (28 out of 41) were bound to hardware. Therefore only those nodes bound to software are listed here.

- kernel nodes bound to software:
  - Sched::doSched, Proc::reset, _ldProcCtxt, _readCwpHiLimit, _schedIsr, _stProcCtxt
- task nodes bound to software:
  - fixdfsi, floor, printDbl, printf, user, cos, divdf3

The resulting weighted cutsets and worst-case execution times per task are listed in Table B.1.

Table B.1: Idle Engine Partition Results

	kern	user	env	cts	fsm	thr	s.a.
|cutset|	n/a	6	1	0	0	1	0
C_i	2862	290436	54412	2562	1363	32855	398

B.3 Generated Problem Data

Eight problems were randomly generated for partitioning. Some statistics describing the generated problems are presented here.

Table B.2: Generated Problem: Nodes per Task

Problem	Average	Minimum	Maximum	Std. Dev.
p100_1	19.4	2	53	18.77
p100_2	22.1	1	49	16.48
p200_1	31.3	1	70	20.86
p200_2	25.8	2	95	25.41
p500_1	30.38	1	103	24.53
p500_2	31.32	3	102	21.58
p1000_1	37.4	1	161	35.72
p1000_2	37.18	1	139	33.04

Table B.3: Generated Problem: Edges per Task

Problem	Average	Minimum	Maximum	Std. Dev.
p100_1	29.1	1	93	32.79
p100_2	30.5	0	75	26.35
p200_1	42.45	0	112	33.87
p200_2	32.9	1	158	39.71
p500_1	34.2	0	142	31.44
p500_2	33.48	2	118	25.31
p1000_1	40.2	0	205	42.31
p1000_2	39.51	0	167	38.70

Table B.4: Generated Problem: Processor Demand per Task

	C_i/T_i per Task		C_i/D_i per Task	
Problem	Average	Std. Dev.	Average	Std. Dev.
p100_1	0.1	1.178e-09	0.1448	0.02194
p100_2	0.1040	0.008620	0.2020	0.09268
p200_1	0.06138	0.02384	0.1383	0.09016
p200_2	0.05476	0.007520	0.1218	0.05962
p500_1	0.02394	0.008624	0.07761	0.09298
p500_2	0.02295	0.006848	0.06823	0.08130
p1000_1	0.01	0	0.01159	0.0009630
p1000_2	0.01132	0.003086	0.02771	0.02471

B.4 Heuristic Gain Results

Detailed results are tabulated here for the heuristic gain experiment conducted in Chapter 7.

Table B.5: Heuristic Gain Results: idle engine

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	0.14	0	0	0	0	0.0002920	0	0
g^χ	0.10	0	0	0	0	4.200e-07	0	0
g^U	0.12	31	0.4657	0.4807	0.01881	8.442e-06	0	0
g^U/LE	0.26	100	0.4657	0.4964	0.01350	1.470e-05	0	0

Table B.6: Heuristic Gain Results: p100_1

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	0.65	0	0	0	0	2.644e-05	0	0
g^χ	4.99	40	0.8984	0.8992	0.001114	1.533e-07	23	23
g^U	0.52	82	0.8984	0.8984	0	3.983e-07	0	0
g^U/LE	0.57	83	0.8984	0.8984	0	2.925e-07	0	0

Table B.7: Heuristic Gain Results: p100_2

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	0.41	0	0	0	0	0.0002534	0	0
g^χ	0.32	3	0.8512	0.8512	0	0	3	3
g^U	0.35	100	0.8512	0.8512	0	0	0	0
g^U/LE	0.37	100	0.8512	0.8512	0	0	0	0

Table B.8: Heuristic Gain Results: p200_1

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	31.18	92	0.7586	0.7833	0.02082	0.0001358	6	0
g^χ	0.68	0	0	0	0	0	0	0
g^U	25.11	92	0.6772	0.7372	0.02347	6.623e-05	0	0
g^U/LE	25.52	98	0.6772	0.7313	0.02316	8.702e-05	0	0

Table B.9: Heuristic Gain Results: p200_2

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	0.69	0	0	0	0	0.0004072	0	0
g^χ	0.63	3	0.8431	0.8480	0.003494	0	0	0
g^U	1.57	100	0.7836	0.7836	0	0	0	0
g^U/LE	1.84	100	0.7836	0.7836	0	0	0	0

Table B.10: Heuristic Gain Results: p500_1

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	5.61	0	0	0	0	0.0001795	0	0
g^χ	2.72	0	0	0	0	2.683e-08	0	0
g^U	97.92	4	0.5538	0.5613	0.006246	7.658e-05	96	0
g^U/LE	120.93	0	0	0	0	8.483e-05	100	0

Table B.11: Heuristic Gain Results: p500_2

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	3.92	0	0	0	0	0.0003413	0	0
g^χ	2.26	5	0.7089	0.7745	0.03280	0	0	0
g^U	2.45	100	0.7064	0.7064	0	0	0	0
g^U/LE	3.14	99	0.7064	0.7064	0	2.783e-08	0	0

Table B.12: Heuristic Gain Results: p1000_1

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	41.16	0	0	0	0	0.0001125	0	0
g^χ	129.31	14	0.9259	0.9270	0.0003177	5.804e-07	63	0
g^U	204.37	68	0.7739	0.8829	0.04015	2.470e-06	28	0
g^U/LE	207.23	52	0.8275	0.9085	0.02619	2.761e-06	44	0

Table B.13: Heuristic Gain Results: p1000_2

Gain	Time	#	U			χ	Repairs	
	(s)	Feas	min	avg	std dev	avg	Attempt	Success
g^c	13.56	0	0	0	0	0.0002376	0	0
g^χ	6.17	1	0.4063	0.4063	0	0	0	0
g^U	7.12	100	0.2445	0.2445	0	1.121e-09	0	0
g^U/LE	9.03	100	0.2445	0.2445	0	1.121e-09	0	0

Appendix C

Glossary of Acronyms

API - Application Programming Interface

ASIC - Application Specific Integrated Circuit

CDFG - Control and Data-Flow Graph

CTS - Continuous Time System

DAG - Directed Acyclic Graph

DES - Discrete Event System

EDF - Earliest Deadline First

FPGA - Field Programmable Gate Array

FSM - Finite State Machine

HDL - Hardware Description Language

IC - Integrated Circuit

ILP - Integer Linear Programming

IR - Internal Representation

IRQ - Interrupt ReQuest

KB - Kilo Byte

MB - Mega Byte

MHz - Mega Hertz

MIPS - Millions of Instructions Per Second

OS - Operating System

NLP - Non-Linear Programming

P - Proportional band controller

PID - Proportional-Integral-Derivative controller

RAM - Random Access Memory

RISC - Reduced Instruction Set Computer

RPM - Rotations Per Minute

RTOS - Real-Time Operating System

Slif - System Level Intermediate Representation

SoC - System on Chip

SoPC - System on Programmable Chip

SRAM - Static Random Access Memory

UART - Universal Asynchronous Receiver Transmitter

VLSI - Very Large Scale Integration